VŠB – Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra aplikované matematiky

Analýza škálovatelnosti a přesnosti paralelních FFT algoritmů při násobení obrovských celých čísel

The analysis of the scalability and accuracy of parallel FFT algorithms for the huge integers multiplication

2016 Ivana Rotterová

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra aplikované matematiky

Zadání diplomové práce

Student: **Bc. Ivana Rotterová**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 1103T031 Výpočetní matematika

Téma: Analýza škálovatelnosti a přesnosti paralelních FFT algoritmů při násobení obrovských celých čísel
The analysis of the scalability and accuracy of parallel FFT algorithms for the huge integers multiplication

Jazyk vypracování: čeština

Zásady pro vypracování:

Násobení obrovských celých n-ciferných čísel může být s využitím rychlé Fourierovy transformace (FFT) provedeno v čase O(n log(n)) namísto časové náročnosti O(n^2) charakteristické pro standardní násobení. Cílem práce je seznámit se s touto technikou založenou na násobení polynomů, jednotlivými algoritmy FFT a jejich open-sourceovými implementacemi a to jak sekvenčními tak paralelními a superpočítačem pro paralelní testování. To se by se mělo týkat hlavně analýzy paralelní škálovatelnosti vlastní implementace FFT (vybrané konkrétní varianty FFT např. s využitím knihovny PETSc) a implementací v open-sourceových knihovnách, tedy toho, jak výpočetní čas závisí na počtu použitých výpočetních jader, a to pro různé počty cifer a různé použité číselné základy.

Seznam doporučené odborné literatury:

[1] http://en.wikipedia.org/wiki/Multiplication_algorithm

[2]
http://www.cs.rug.nl/~ando/pdfs/Ando_Emerencia_multiplying_huge_integers_using_fourier_transforms_

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Horák, Ph.D.**

Datum zadání: 01.09.2014
Datum odevzdání: 29.04.2016

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA

doc. RNDr. Jiří Bouchala, Ph.D.
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

V Ostravě 29. dubna 2016 .....

Tady bude prohlaseni spolupracujicich pravnicke nebo fyzicke osoby.

Ráda bych na tomto místě poděkovala panu Ing. Davidu Horákovi, Ph.D. za jeho povzbuzující a motivující přístup a cenné připomínky k vypracování mé diplomové práce.

Rovněž bych ráda poděkovala své rodině za trpělivost a podporu, kterou mi po celou dobu poskytovala.

Abstrakt

Cílem práce je seznámení s technikou násobení obrovských celých čísel pomocí polynomů, ilustrace převodů čísel na polynomy s různými číselnými základy a FFT násobení, které je dále porovnáváno s klasickým násobením. Čtenář je seznámen se základy FFT a základními algoritmy pro výpočet FFT, Radix-2, Radix-4, Split-Radix. Dále práce srovnává různé implementace FFT s jejich aplikací pro násobení obrovských čísel včetně škálovatelnosti pomocí paralelních FFT knihoven jak se sdílenou, tak distribuovanou pamětí, testovaných na ostravském superpočítači. Smyslem této práce je poukázat na alternativní možnosti násobení obrovských čísel nejen proto, že při klasickém násobení jsme limitování počtem cifer z důvodu omezené paměti, ale také jsme schopni dosáhnout výsledků mnohem efektivněji.

Klíčová slova: DFT, FFT, IFFT, Radix-2, Radix-4, Split-Radix, MPI, OpenMP, násobení, superpočítač

Abstract

This Thesis is focused on a technigue of multiplication of huge integers using polynomials, conversion of numbers to polynomials with different numerical bases and FFT multiplication, which is also compared with the normal multiplication. A reader will know FFT fundations and algorithms Radix-2, Radix-4, Split-Radix. There is also a comparison of various FFT implementation for multiplication of huge numbers including a scalability by parallel FFT libraries using shared and distributed memory and tested on Ostrava supercomputer. This Thesis meaning has been to point out alternative ways how to multiply huge integers not only because we are limited by number of digits in case of normal multiplication but also we are able to achieve results more effective.

Key Words: DFT, FFT, IFFT, Radix-2, Radix-4, Split-Radix, MPI, OpenMP, multiplication, supercomputer

Obsah

Seznam použitých zkratek a symbolů

FT	–	Fourier Transformation
DFT	–	Discrete Fourier Transformation
FFT	–	Fast Fourier Transformation
IFFT	–	Inverse Fast Fourier Transformation
OpenMP	–	Open Multi-Processing
MPI	–	Message Passing Interface

Seznam obrázků

Seznam tabulek

Obsah

Seznam výpisů zdrojového kódu

1 Úvod/Historie FT a FFT

Historie FFT - 1805

Dle záznamů objevených z pozůstalosti C. F. Gausse v roce 1866 je historie **Rychlé Fourierovy transformace (FFT - Fast Fourier Transformation)** datována již okolo roku 1805. Pro zbytek světa však byla utajena ještě dalších 160 let. Gauss tuto svou práci nikdy nepublikoval, využíval ji pro své interní účely při určování oběžných drah asteroidů.

Při tomto výzkumu vyvinul to, co je dnes známo jako **Diskrétní Fourierova Transformace (DFT)**, kterou Fourier publikoval až v roce 1822.

Pro výpočet DFT rozdělil původní DFT *na dvě o polovičních délkách a počítal zvlášť sudé a liché členy* (detail později u dvojice Cool-Tukey).

Takto využívá metodu, jejíž složitost je $O(N \log N)$, pro interpolaci oběžné dráhy nebeských těles.

Historie FFT - 1828 - 1958

V následujících letech byly vyvinuty různé metody pro výpočet DFT, žádná z nich však nebyla natolik obecná jako Gauss/Cool-Tukey.

Principal Discoveries of Efficient Methods of Computing the DFT

Researcher(s)	Date	Sequence Lengths	Number of DFT Values	Application
C. F. Gauss [10]	1805	Any composite integer	All	Interpolation of orbits of celestial bodies
F. Carlini [28]	1828	12	—	Harmonic analysis of barometric pressure
A. Smith [25]	1846	4, 8, 16, 32	5 or 9	Correcting deviations in compasses on ships
J. D. Everett [23]	1860	12	5	Modeling underground temperature deviations
C. Runge [7]	1903	$2^n k$	All	Harmonic analysis of functions
K. Stumpff [16]	1939	$2^n k$, $3^n k$	All	Harmonic analysis of functions
Danielson and Lanczos [5]	1942	2^n	All	X-ray diffraction in crystals
L. H. Thomas [13]	1948	Any integer with relatively prime factors	All	Harmonic analysis of functions
I. J. Good [3]	1958	Any integer with relatively prime factors	All	Harmonic analysis of functions
Cooley and Tukey [1]	1965	Any composite integer	All	Harmonic analysis of functions
S. Winograd [14]	1976	Any integer with relatively prime factors	All	Use of complexity theory for harmonic analysis

Obrázek 1: Vývoj metod DFT/FFT [4]

Historie FFT - 1965

V tomto období dvojice matematiků **Cooley a Tukey** ([4], [5], [6]) znovu-objevují algoritmus pro výpočet DFT založen na stejných principech jako je původní **Gaussův.**

Cílem je využití při odhalování jaderných testů Sovětského svazu.

Algoritmus funguje pro složená čísla, nikoliv prvočísla, což vyplývá z podstaty rekurzivního dělení DFT na menší určité složené velikosti

$$N = N_1 \times N_2.$$

Algoritmy jsou založeny na rekurzi a při implementaci jsou z důvodu paměťové náročnosti obvykle převáděny na iterativní. Patří zde algoritmy **Radix-2 a Radix-4**.

Od DFT k FFT

Diskrétní Fourierova Transformace (DFT) transformuje posloupnost délky N na jinou posloupnost délky N:

$$x(n) \xrightarrow{DFT} X(k), \qquad X(k) = \sum_{n=0}^{N-1} x(n) e^{-i\frac{2\pi}{N}kn}.$$

Tato transformace má řadu využití:

- *spektrální analýza* - převod signálu z časové do frekvenční oblasti,
- *komprese dat* - ztrátové komprese (JPEG),
- *parciální diferenciální rovnice (PDR)* - transformace na lépe řešitelné algebraické rovnice,
- *násobení polynomů* a další.

Pro výpočet transformace velikosti N je třeba provést N^2 komplexních součinů, mluvíme tedy o složitosti $O(N^2)$, kdy doba trvání průběhu se zvyšuje kvadraticky, jedná se tedy o kvadratickou složitost.

DFT je početně velmi náročná metoda, založena na konečných řadách a goniometrických funkcích s časovou složitostí $O(N^2)$. Pro vyšší hodnoty N se výpočetní čas zvyšuje velmi rychle, což je nežádoucí. Tato skutečnost vytváří nutnost zamýšlet se nad možností optimalizace, tedy zvýšit výkon, vytvořit úspornější algoritmy, a tak vzniká Fast Fourier Transformation (FFT), tj.

algoritmy mnohem výkonnější, s nižší časovou složitostí $O(N \log N)$, např. algoritmy **Radix-2, Radix-4 a Split Radix.**

První a velmi významný FFT algoritmus byl vyvinut dvojicí **Cooley a Tukey**.

2 Metody FFT

2.1 Klasické DFT - implementace

Klasický algoritmus DFT nenabízí příliš možností optimalizace. Pro každou hodnotu musíme projít celým polem. Vzorec využívá hodnoty vnitřního cyklu a kromě $\frac{2\pi}{N}$ nelze nic dalšího předpočítat.

DFT je definována pomocí formule:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-i\frac{2\pi}{N}nk}.$$

V klasických programovacích jazycích, např. C/C++, nelze pracovat s takovými matematickými operacemi bez speciálních knihoven, využijeme proto rovnosti

$$e^{ix} = \cos x + i \sin x$$

a výše zmíněný tvar vyjadříme následovně:

$$X(k) = \sum_{n=0}^{N-1} x(n) \left(\cos\left(-\frac{2\pi}{N}nk\right) + i \sin\left(-\frac{2\pi}{N}nk\right)\right).$$

Po úpravě pak vzhledem ke znalostem sudých a lichých funkcí získáme tvar:

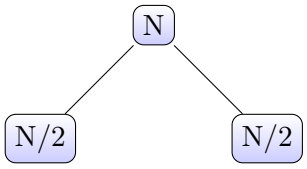
$$X(k) = \sum_{n=0}^{N-1} x(n) \left(\cos\left(\frac{2\pi}{N}nk\right) - i \sin\left(\frac{2\pi}{N}nk\right)\right),$$

$$X(k) = \sum_{n=0}^{N-1} x(n) \cos\left(\frac{2\pi}{N}nk\right) - i \sum_{n=0}^{N-1} x(n) \sin\left(\frac{2\pi}{N}nk\right).$$

2.2 Radix-2 (RAD2)

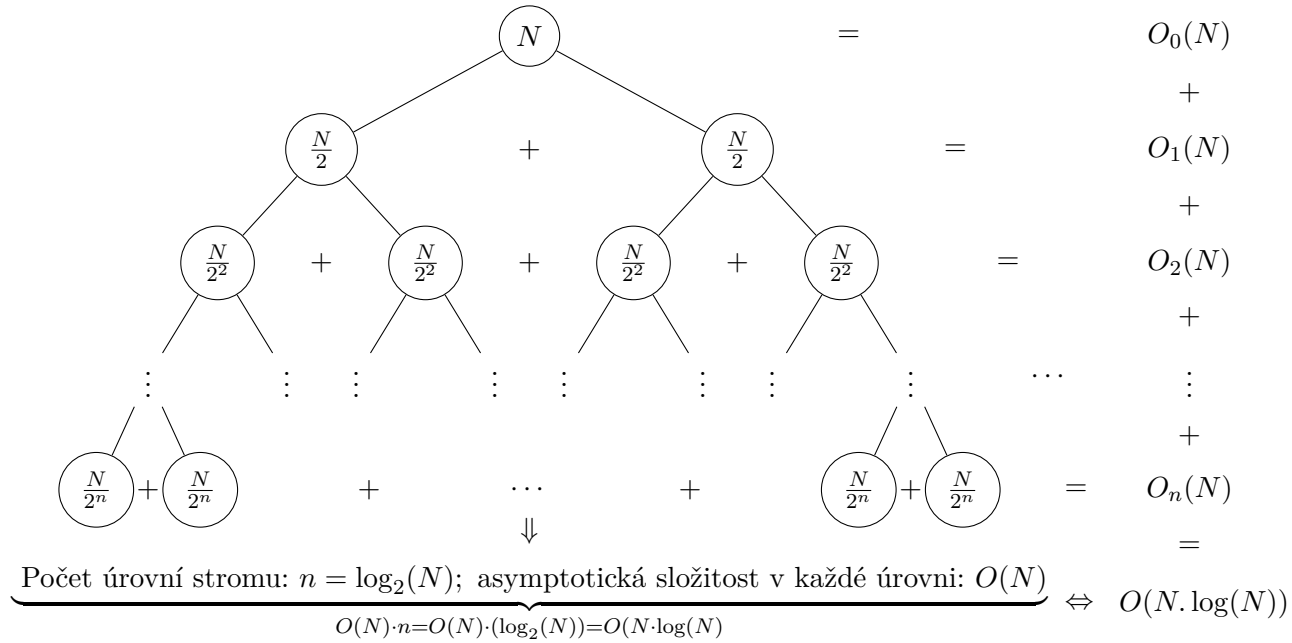
Radix-2 ([1]) je nejznámějším algoritmem pro délku záznamu (počet vzorků), která odpovídá mocnině dvou, tj. $N = 2^n$. Byl vyvinut dvojicí **Cooley a Tukey**. Překládá se česky jako algoritmus decimování v čase (DIT).

Využívá symetrie a komplexní sdruženosti kolem středu. DFT rozklad je rozdělen na dvě $N/2$ části, sudé $2n$ a liché $2n + 1$ členy, a jejich kombinací vzniká výsledek.



Obrázek 2: DFT rozklad

Tato myšlenka pak může být prováděna rekurzivně, jelikož předpokladem je $N = 2^n$ a je-li $N > 1$, dochází k dělení pole prvků tak dlouho, dokud nezískáme jednotlivé samostatné prvky.



Obrázek 3: DFT rozklad - rekurzivní dělení

Následně získáme dvě části, a to součet přes sudé $2n$ a přes liché $2n + 1$ indexy.

Pro výpočet pak stačí spočítat reálnou část a pro imaginární pouze obrátit znaménko.

$$X(k) = \underbrace{\sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-i\frac{2\pi}{N}(2n)k}}_{\text{sudé}} + \underbrace{\sum_{n=0}^{\frac{N}{2}-1} x(2n+1)e^{-i\frac{2\pi}{N}(2n+1)k}}_{\text{liché}},$$

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-i\frac{2\pi}{N}(2n)k} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)e^{-i\frac{2\pi}{N}(2n)k - i\frac{2\pi}{N}(1)k},$$

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-i\frac{2\pi}{N}(2n)k} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)e^{-i\frac{2\pi}{N}(2n)k}.e^{-i\frac{2\pi}{N}k},$$

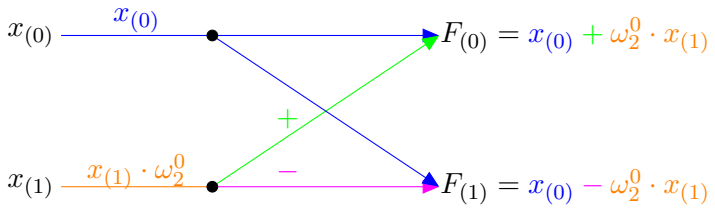
$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-i\frac{2\pi}{N}(2n)k} + e^{-i\frac{2\pi}{N}k}. \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)e^{-i\frac{2\pi}{N}(2n)k},$$

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-i\frac{4\pi}{N}nk} + e^{-i\frac{2\pi}{N}k}. \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)e^{-i\frac{4\pi}{N}nk}.$$

Druhá polovina bude identická s opačným znaménkem

$$X\left(\frac{N}{2}+k\right) = \sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-i\frac{4\pi}{N}nk} - e^{-i\frac{2\pi}{N}k}. \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)e^{-i\frac{4\pi}{N}nk}.$$

Označme si $\omega_N^k = e^{-i\frac{2\pi}{N}k}$ a nazveme ho **"twiddle faktor"**. A mějme dvojici vzorků $x(0), x(1)$, k nimž dojdeme rekurzivním dělením.



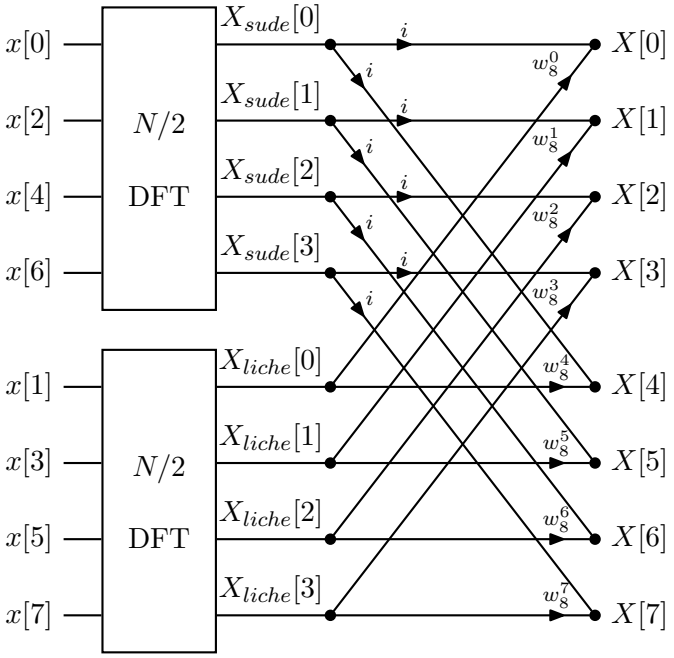
Obrázek 4: Dvouprvkový motýlek

"Motýlkem" nazveme diagram znázorňující postup výpočtu, jehož produktem je dvojice koeficientů DFT. Tato struktura odpovídá i hardwarovému zapojení, kdy pro samotný výpočet musíme propojit více **motýlků**.

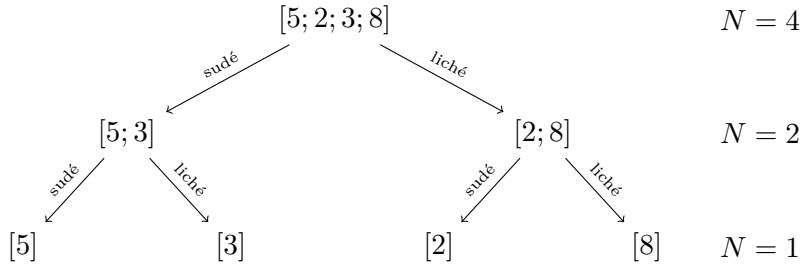
Twiddle faktor ω_N^k pak nese informaci o velikosti úhlu, resp. $x(n)\omega_N^k$ vyjadřuje rotaci $x(n)$ v komplexních souřadnicích a můžeme ho nazvat **faktorem míry pootočení**.

Složitost tohoto algoritmu je $5N \log N$. Paměťově se jedná o nejméně náročný algoritmus (iterativní implementace).

2.2.0.1 Příklad: Mějme číslo 5238, při výpočtu FFT obrazu budeme postupovat následovně: dokud $N \geq 2 \Rightarrow$ budeme dělit pole prvků na dvě poloviny, přičemž rozlišujeme sudé a liché pořadí prvků a indexujeme od nuly, $N = 0, 1, 2, 3$:



Obrázek 5: Sekce motýlků pro 8 prvků



Obrázek 6: Postup výpočtu - DFT rozklad

V prvním kroku $N = 4 \Rightarrow$ rozdělíme čtyřprvkové pole na dvě pole poloviční velikosti:

- prvky sudého pořadí (nultý a druhý): [5; 3]
- prvky lichého pořadí (první a třetí): [2; 8]

Ve druhém kroku $N = 2 \Rightarrow$ pole (tentokrát máme již dvě) rozdělíme opět na dvě poloviny (v obou případech):

- prvek sudého pořadí: [5]
- prvek lichého pořadí: [3]
- prvek sudého pořadí: [2]
- prvek lichého pořadí: [8]

Ve třetím kroku již není co dělit a ukládáme do proměnné konkrétní číslo jako **sudé/liché prvky**:

sudý prvek = [5]

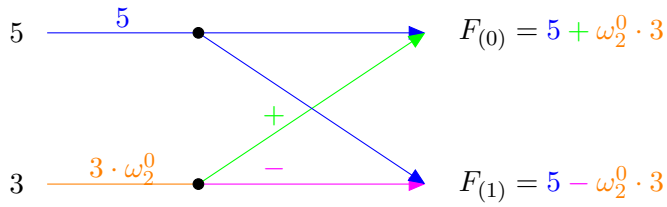
lichý prvek = [3]

Dříve zmíněný **twiddle faktor** $\omega_N^k = e^{-i\frac{2\pi k}{N}}$ počítáme vždy pro patřičné $k = 0, ..., \frac{N}{2} - 1$, přičemž v prvním zpětném kroku, kdy máme jeden prvek sudého a jeden lichého pořadí, platí: $N = 2 \Rightarrow k = 0$:

$$\omega_N^k = e^{-i.0} = 1$$

$\omega^k_{N\text{Liché číslo}}$[1] $= \omega_N^k$. lichý prvek $= 1.3 = 3$

$y =$ transformované sudé číslo $\pm\ \omega^k_{N\text{Liché číslo}} = 5 \pm 3 = [5+3; 5-3] = \underline{\underline{[8;2]}}$.



Obrázek 7: Ilustrace motýlka z našeho příkladu - levá větev

Totéž provedeme pro druhou větev a postupně získáme (ω_N^k je stejné):

sudý prvek = [2]

lichý prvek = [8]

$\omega^k_{N\text{Liché číslo}} = \omega_N^k$. lichý prvek $= 1.8 = 8$

$y =$ transformované sudé číslo $\pm\ \omega^k_{N\text{Liché číslo}} = 2 \pm 8 = [2+8; 2-8] = \underline{\underline{[10;-6]}}$.

V dalším zpětném chodu spojujeme dvě pole o dvou prvcích, tj. $N = 4 \Rightarrow k = 0, 1$. Tyto hodnoty dosadíme postupně za k a určíme ω_N^k:

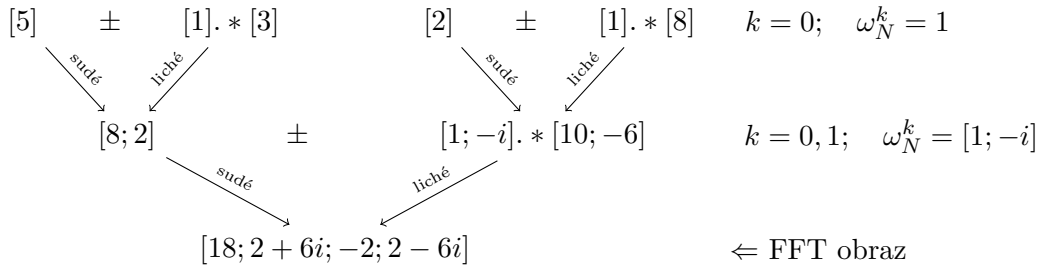
$$\omega_N^k = e^{-i\frac{2\pi k}{N}} = [e^{-i\frac{2\pi.0}{N}}; e^{-i\frac{2\pi.1}{N}}] = [e^0; e^{-i\frac{2\pi}{4}}] = [1; e^{-i\frac{\pi}{2}}] = [1; cos\frac{\pi}{2} - i.sin\frac{\pi}{2}] = [1; 0 - i.1] = \underline{\underline{[1;-i]}}$$

$\omega^k_{N\text{Liché číslo}} = \omega_N^k$. lichý prvek $= [1, -i]. * [10; -6] = \underline{\underline{[10; 6i]}}$.

[1] Zavedeme si $\omega^k_{N\text{Liché číslo}}$, což bude lichý prvek přenásobený ω_N^k

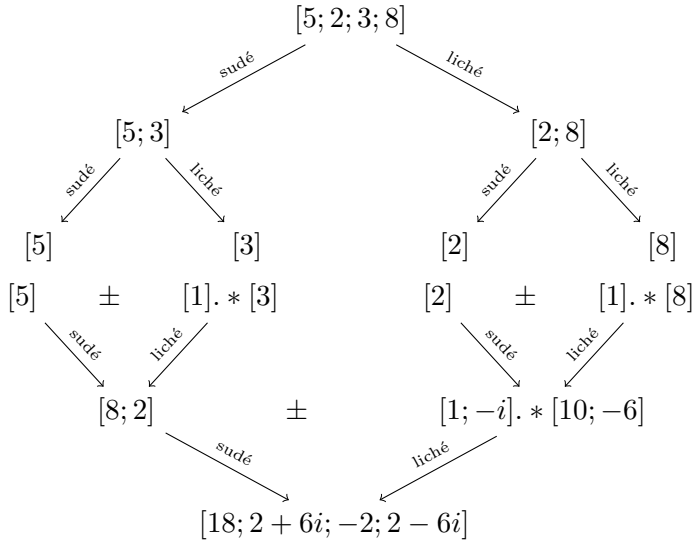
y = transformované sudé číslo $\pm\ \omega_{N\,\text{Liché číslo}}^{k} = [8;2] \pm [10;6i] = [[8;2]+[10;6i];[8;2]-[10;6i]] =$
$= \underline{\underline{[18;2+6i;-2;2-6i]}}$.

A právě tato hodnota y představuje výsledek motýlků Radix2, blíže ukázka následujícího kódu rekurzivní verze v další podkapitole.



Obrázek 8: Postup výpočtu - FFT

Celý rekurzivní postup pak vidíme na Obrázku 9[2]:



Obrázek 9: Rekurzivní postup výpočtu FFT

[2] Všimněme si, že twiddle faktorem ω_N^k násobíme vždy pravý uzel stromu.

2.3 Radix-2 rekurzivní

Rekurzivní implementace je **jednodušší, ovšem náročná na paměť, neboť vytváří mnoho pomocných polí při každém volání** (exponenciální prostorová složitost):

```
1  function y = radix2_rekurzivni(x)
2
3  % pocet prvku v rekurzivnim kroku
4  N = length(x);
5
6  % dokud nemame pole rozdelene na jeden sudy a jeden lichy prvek (motylek),
7  % tak provadime deleni, jinak vracime hodnotu tohoto prvku
8  if N == 1
9      y = x;
10 else
11     % rozdeleni vstup na dve N/2 posloupnosti ze sudych a lichych prvku
12     liche = x(1:2:N-1); % liche prvky: x[0], x[2], x[4], ..., x[N-1].
13     sude  = x(2:2:N);   % sude prvky : x[1], x[3], x[5], ..., x[N].
14
15     % rekurzivne vola radix2
16     transformovaneLiche = radix2_rekurzivni( liche );
17     transformovaneSude = radix2_rekurzivni(sude);
18
19     % DFT vypocitava pro polovinu prvku
20     k = 0:(N/2)-1;
21
22     % spocitame twidle factor pro vsechny k
23     W = exp(-2i*pi*k/N);
24
25     % Motylky pocitame pouze pro prvni polovinu prvku: Wnk (twidle factor) pronasobime s lichymi
           hodnotami (x1)
26     % vyslednou hodnotu pak pricteme a odecteme od sude hodnoty (x0)
27     WnkLiche = W .* transformovaneLiche;
28
29     % k-ty a k+(n/2) prvek vysledneho pole, vyuzitim symetricnosti se snizil pocet nasobeni
30     y = [transformovaneSude + WnkLiche, transformovaneSude - WnkLiche];
31 end
```

Výpis 1: Radix-2 rekurzivní: Matlab kód

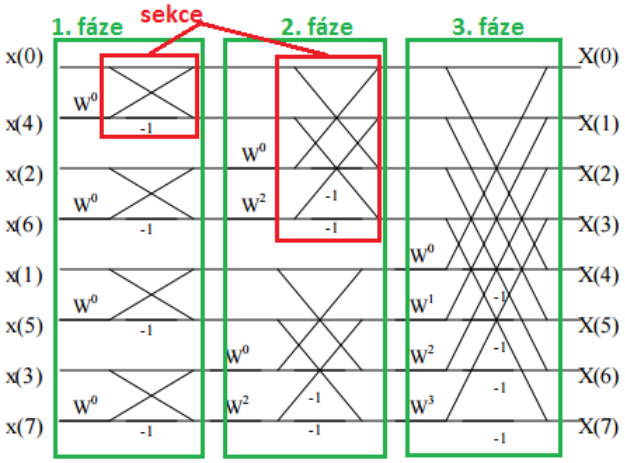
2.4 Radix-2 iterativní

Oproti předchozí implementaci, která je paměťově velmi náročná, se můžeme vyhnout rekurzivnímu volání pomocí tzv. **přehození pořadí bitů** (v angličtině často jako **bit reversal** nebo **bit revorder**) a na krátké ukázce si předvedeme, co vlastně to obrácení pořadí znamená:

Původní pozice		Nová pozice	
Desítkové	Binární	Binární	Desítkové
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Tabulka 2: Přeuspořádání pomocí bitové inverze

Převedeme indexy $0, 1, ..., N-1$ na jejich binární reprezentaci a každý takový bitový záznam obrátíme, viz tabulka 2. Získáme tak nové pořadí prvků, kde každé dva prvky, které je třeba spárovat, již stojí vedle sebe. Jsou to přesně tytéž dvojice prvků, k nimž jsme došly složitým dělením pomocí rekurze. Využití možností obrátit pořadí prvků nám umožňuje vyhnout se rekurzivnímu dělení na jednotlivé prvky, které s sebou přinášelo nutnost alokovat velké množství paměti.



Obrázek 10: 8-bodový Radix-2 - FFT DIT

Popis algoritmu

Algoritmus se podobá rekurzivní variantě. Výpočet je rozdělen na $\log_2(N)$ fází. Ty odpovídají hloubce stromu u rekurzivní metody. Každá fáze je pak rozdělena na sekce s určitým počtem motýlků [3] a počet těchto sekcí odpovídá počtu uzlů stromu v dané fázi. Jako pomocnou pro-

[3] Na obrázku 8-bodový Radix-2 - FFT DIT představuje výpočet, kde v první fázi obsahuje každá sekce jednoho motýlka, druhá fáze má motýlky dva a ve třetí fázi jsou čtyři motýlci

měnnou pro velikost vektoru N v dané fázi si zavedeme lokální proměnnou M, jež odpovídá velikosti vektoru v jednom uzlu stromu. Pro každou sekci spočítáme její motýlky.

```
function y = radix2_iterativni_DIT(x)

N = length(x);              % pocet prvku vstupniho vektoru (konstatni pro kazdou fazi vypoctu)
pocetFazi = log2(N);        % pocet fazi = hloubka stromu u rekurzivni varianty
x = x(bitrevorder(1:N));    % pretridime prvky pomoci bit-reversal.

% samotny vypocet, odpovida druhe casti rekurzivni varianty od listu ke koreni stromu
for faze = 1:pocetFazi
    M = 2^faze;             % pocet prvku na sekci v dane fazi (velikost N v danem uzlu stromu)
    pocetSekci = N/M;       % pocet sekci v dane fazi

    for sekce=0:pocetSekci-1        % pro kazdou sekci spocitame jednotlive motylky
        for k = 0:(M/2)-1           % posun uhlu twiddle faktoru

            n0 = (sekce*M) + k + 1; % index sudeho prvku
            n1 = n0 + M/2;          % index licheho prvku

            W = exp(-2i*pi*k/M);    % spocitame twidle faktor W = e^(-i*2pi*k/M)

            x0 = x(n0);             % sudy prvek
            x1 = x(n1)*W;           % lichy prvek pronasobime Wnk

            x(n0) = x0 + x1;
            x(n1) = x0 - x1;
        end
    end
end
y = x;
```

Výpis 2: Radix-2 iterativní DIT: Matlab kód

2.4.1 Radix-2 iterativní DIT - optimalizace

Tento algoritmus je možné optimalizovat různými způsoby. Např. přehozením pořadí cyklů nebo předpočítáním twiddle faktoru. V jazycích, které nemají implementovánu práci s komplexními čísly a využíváme Eulerův vzorec, se také dá výpočet urychlit předpočítáním hodnot **sin** a **cos** a následně je použít pomocí indexů. Ukázkovou úpravou se algoritmus zrychlil o cca 15%.

```
function y = radix2_iterativni_DIT_optimalizovane(x)

N = length(x);
pocetFazi = log2(N);
x = x( bitrevorder (1:N));

for faze = 1:pocetFazi
    M = 2^faze;
    M_pul = M/2;                    % hodnotu M/2 si predpocteme
    pocetSekci = N/M;

    % prehozenim poradi cyklu usetrime pocet opakovaneho vypoctu twidle faktoru
    for k = 0:M_pul-1
        % twidle faktor predpocteme. Teoreticky by sla cast vytknout i pred prvni cyklus a zde
            pronasobit pouze k-hodnotou, tj. vytknout W = exp(-2i*pi/M);
        % V cyklu pak pocitat Wk = W.^k, ale realne vysledky v Matlabu jsou horsi, nez kdyz se vypocet
            provede najednou.
        W = exp(-2i*pi*k/M);

        for sekce=0:pocetSekci-1
            n0 = (sekce*M) + k + 1; % index sudeho prvku
            n1 = n0 + M_pul;        % index licheho prvku

            x0 = x(n0);             % sudy prvek
            x1 = x(n1)*W;           % lichy prvek pronasobime twidle faktorem

            x(n0) = x0+x1;
            x(n1) = x0-x1;
        end
    end
end
y = x;
```

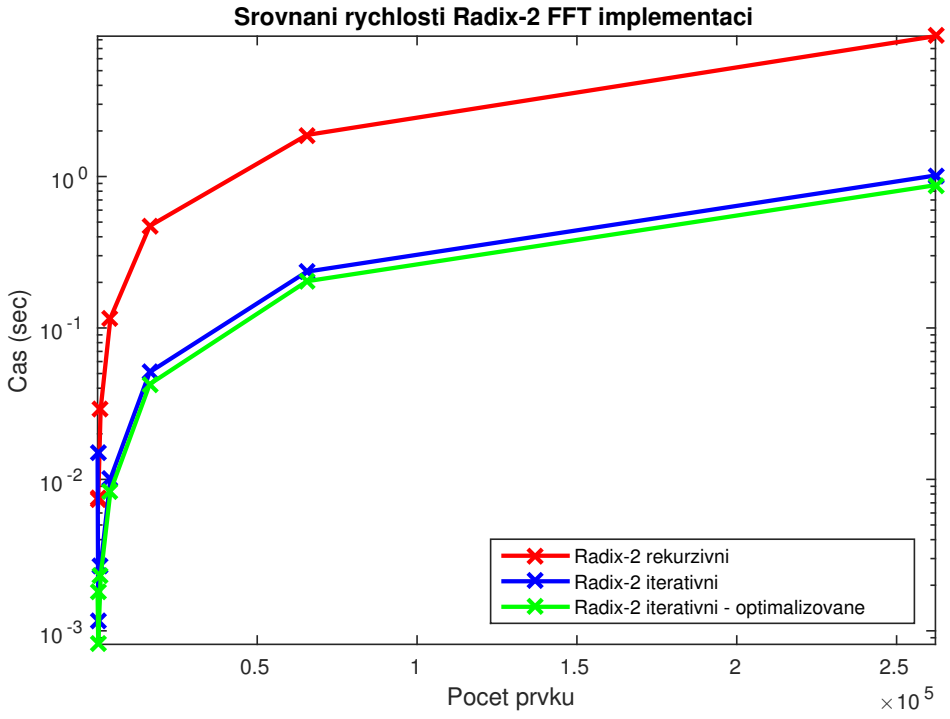
Výpis 3: Radix-2 iterativní (optimalizovaný): Matlab kód

Z následujícího měření založeného na sledování počtu vzorků[4], což charakterizuje tabulka 3 a dále graf, si můžeme udělat představu o rychlosti zpracování dat zmíněných algoritmů.

Radix-2 / počet vzorků	1024	4096	16 384	65 536	262 144
Rekurzivní	0.0292	0.1165	0.4714	1.8811	8.4500
Iterativní	0.0027	0.0101	0.0511	0.2354	1.0157
Iterativní optimalizovaný	0.0023	0.0084	0.0426	0.2035	0.8766

Tabulka 3: Srovnání implementací Radix-2 FFT (čas v sekundách)

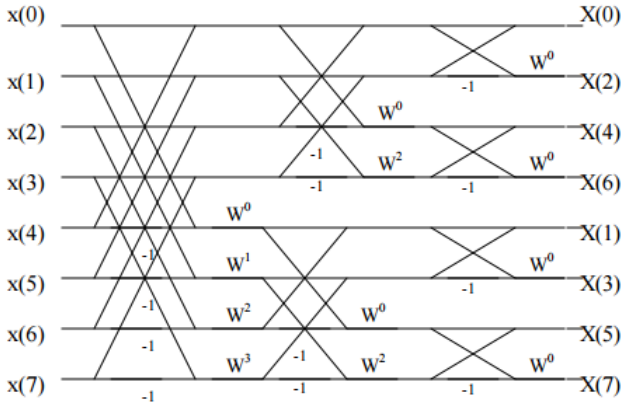
[4]Jedná se o délku čísla, tj. počet míst.



Obrázek 11: Srovnání implementací Radix-2 FFT

2.4.2 Radix-2 iterativní DIF

Ve srovnání s DIT variantou postupujeme u DIF obráceně, začínáme s jednou sekcí a v každé další fázi výpočet dělíme na poloviční sekce, až skončíme s jedním motýlkem na sekci. Tento postup mění pořadí prvku tak, že bit reversal je použit až na závěr výpočtu, narozdíl od DIT.



Obrázek 12: 8-bodový Radix-2 - FFT DIF[7]

```
function y = radix2_iterativni_DIF(x)

% Varianta DIF iterativniho vypoctu. Narozdil od DIT postupujeme obracene, zaciname s jednou sekci a v
    kazde dalsi fazi vypocet delime na polovicni sekce,% az skoncime s jednim motylkem na sekci.
    Tento postup meni poradi prvku tak, ze bit-reversal pouzijeme az na zaver vypoctu, narozdil od
    DIT.

N = length(x);
M = N;                      % zaciname jednou sekci, ve ktere se zpracuji vsechny DFT
pocetFazi = log2(N);

for faze = 1:pocetFazi
    pocetSekci = N/M;           % spocitame na kolik sekci bude rozdelena dana faze
    for sekce = 0:(pocetSekci-1) % pro kazdou sekci spocitame jednotlive motylky
        for k=0:(M/2)-1;

            % vypocteme indexy prvku, ze kterych budeme pocitat motylka
            n0 = (sekce*M) + k + 1;
            n1 = n0 + M/2;

            % spocitame twidle faktor W = e^(-i*2pi*k/M), kde M odpovida N prvku v dane sekci
            W = exp(-2i*pi*k/M);

            % vypocet motylka je odlisny oproti DIT variante
            x0 = x(n0);
            x1 = x(n1);

            x(n0) = x0 + x1;
            x(n1) = (x0 - x1)*W;
        end
    end

    % v kazde fazi se se pocet prvku na sekci snizuje na polovinu
    M = M / 2;
end

% na zaver vypoctu pretridime prvky pomoci bit-reversal
y = x( bitrevorder (1:N));
```

Výpis 4: Radix-2 iterativní DIF: Matlab kód

2.5 Radix-2 versus DFT

Myšlenka Radix-2 je jednoduchá, přitom velice účinná. Snahy o ještě lepší výsledky, než pouhou optimalizaci, vedly ke vzniku novějších a efektivnějších algoritmů.

2.6 Radix-4 (RAD4)

Radix-4 je založen na podobném principu jako Radix-2, ale namísto dvou vstupů při použití Radix-2 pracuje se čtyřmi vstupy, má složitější motýlky a stejně tak jako iterativní Radix-2 využíval přeuspořádání pomocí bitové inverze, tak i iterativní Radix-4 dokáže dělit prvky pomocí tzv. **digit reverse**. Rozděluje vzorky na čtveřice, tudíž počet vzorků musí odpovídat mocnině čtyř, tj. $N = 4^m$.

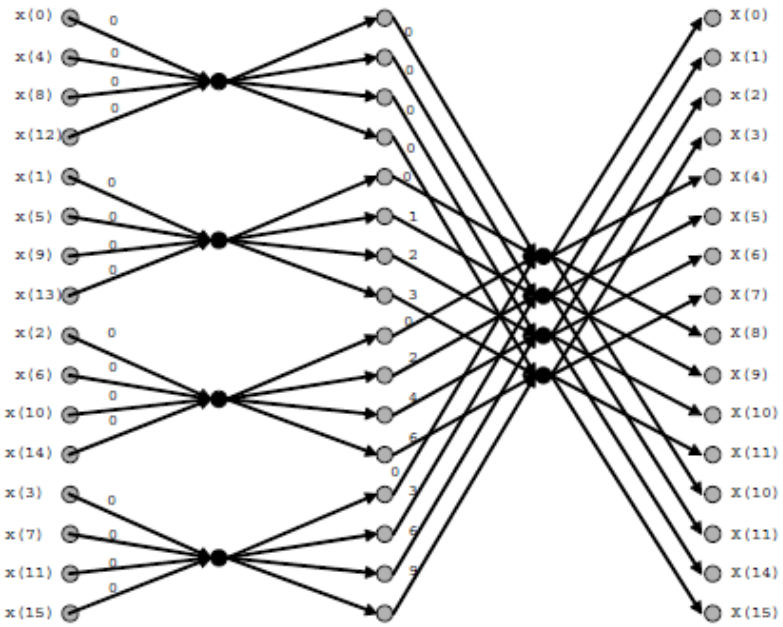
Původní pozice		Nová pozice				
Desítkový zápis	Základ 4	Převrácený základ 4	Nový index desítk. zapsaný	Základ 2	Převrácený základ 2	Umístění prvku
0	00	00	0	0000	0000	0
1	01	10	4	0001	1000	8
2	02	20	8	0010	0100	4
3	03	30	12	0011	1100	12
4	10	01	1	0100	0010	2
5	11	11	5	0101	1010	10
6	12	21	9	0110	0110	6
7	13	31	13	0111	1110	14
8	20	02	2	1000	0001	1
9	21	12	6	1001	1001	9
10	22	22	10	1010	0101	5
11	23	32	14	1011	1101	13
12	30	03	3	1100	0011	3
13	31	13	7	1101	1011	11
14	32	23	11	1110	0111	7
15	33	33	15	1111	1111	15

Tabulka 4: Přeuspořádání pomocí bitové inverze [10]

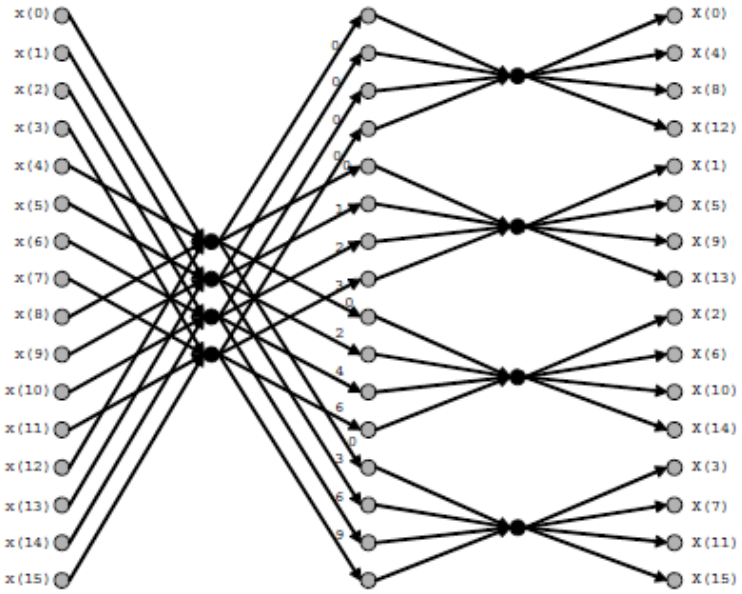
Algoritmus je o 15% rychlejší než Radix-2 a jeho složitost je $4,25N \log N$. Je definován vztahem (více zde [8]):

$$X(k) = \sum_{n=0}^{\frac{N}{4}-1} x(n).\omega_N^{kn} + \sum_{n=\frac{N}{4}}^{\frac{N}{2}-1} x(n).\omega_N^{kn} + \sum_{n=\frac{N}{2}}^{\frac{3N}{4}-1} x(n).\omega_N^{kn} + \sum_{n=\frac{3N}{4}}^{N-1} x(n).\omega_N^{kn}.$$

A pro ilustraci se můžeme podívat, jak vypadá motýlek pro 16 členů při vatiantě DIT a DIF *(DIF je popsán v následující kapitole)*:



Obrázek 13: 16-bodový Radix-4 - FFT DIT[9]



Obrázek 14: 16-bodový Radix-4 - FFT DIF[9]

2.6.1 Radix-4 rekurzivní

```
function y = radix4_rekurzivni(x)
% Radix-4 FFT, rekurzivni verze

% % pocet prvku v rekurzivnim kroku
N = length(x);

if (N == 1)
    % pro N = 1 vracime hodnotu prvku, strom je jiz rozdelen do posledni
    % faze a zaciname pocitat jednotlive transformace
    y = x;
else
    % vstupni vektor rozdelime na 4 posloupnosti, pozn. Matlab indexuje od 1
    a0 = x(1:4:N-3); % N/4 prvky: x[0], x[4], x[8], ..., x[N-3].
    a1 = x(2:4:N-2); % N/4 prvky: x[1], x[5], x[9], ..., x[N-2].
    a2 = x(3:4:N-1); % N/4 prvky: x[2], x[6], x[10], ..., x[N-1].
    a3 = x(4:4:N);   % N/4 prvky: x[3], x[7], x[11], ..., x[N].

    % Rekurzivne vola Radix-4, A0..A3 jsou jiz FFT obrazy vektoru a0..a3 z predchoziho rekurzivniho
        kroku
    A0 = radix4_rekurzivni(a0);
    A1 = radix4_rekurzivni(a1);
    A2 = radix4_rekurzivni(a2);
    A3 = radix4_rekurzivni(a3);

    % predpocitame cast twidle faktoru pro vsechny k
    k = 0:(N/4)-1;
    W = exp(-2i*pi*k/N);

    % kazdy prvek pronasobime twidle faktorem
    A0 = A0.*W.^0;
    A1 = A1.*W.^1;
    A2 = A2.*W.^2;
    A3 = A3.*W.^3;

    % aplikujeme souctovy vzorec
    A = A0 +  A1 + A2 +  A3;
    B = A0 - 1i*A1 - A2 + 1i*A3;
    C = A0 -  A1 + A2 -  A3;
    D = A0 + 1i*A1 - A2 - 1i*A3;

    y = [A B C D];
end;
```

Výpis 5: Radix-4 rekurzivní: Matlab kód

2.6.2 Radix-4 iterativní DIT

```
function y = radix4_iterativni_DIT(x)

N = length(x);                % pocet prvku vstupniho vektoru
pocetFazi = log(N)/log(4); % pocet fazi = hloubka stromu u rekurzivni varianty

% preusporadame prvky pomoci bit-reversal (nutno otacet 4 bity, oproti radix-2)
x = digitrevorder(x,4);

% vypocet je rozdelen na faze, pocet fazi = hloubka stromu u rekurzivni varianty
for faze=1:pocetFazi

    % velikost N se pro kazdou sekci snizuje,
    % pro velikost vektoru v dane fazi zavedeme nove oznaceni M.
    M = 4^faze;

   % pocet sekci v dane fazi
    pocetSekci = N/M;

    for sekce = 0:pocetSekci-1; % pro kazdou sekci spocitame jednotlive motylky
        for k = 0:M/4-1;        % posun uhlu twidle faktoru

            % vypocteme indexy prvku, ze kterych budeme pocitat motylka
            % napr. motylek = x[0] x[4] x[8] x[12] (v Matlabu indexy [1 5 9 13])
            n = (sekce*M) + 1 + k + M/4.*(0:3);

            % hodnoty pronasobime twiddle faktorem W = e^(-i*2pi*k/M)
            % s patricnym posunem 0..3, viz. vzorec
            x0 = x(n(1)) * exp(-2i*pi*0*k/M);
            x1 = x(n(2)) * exp(-2i*pi*1*k/M);
            x2 = x(n(3)) * exp(-2i*pi*2*k/M);
            x3 = x(n(4)) * exp(-2i*pi*3*k/M);

            % aplikujeme souctovy vzorec a vysledky ulozime do puvodniho pole
            x(n(1)) = x0 +   x1 + x2 +   x3;
            x(n(2)) = x0 - 1i*x1 - x2 + 1i*x3;
            x(n(3)) = x0 -   x1 + x2 -   x3;
            x(n(4)) = x0 + 1i*x1 - x2 - 1i*x3;

        end;
    end;
end;
y=x;
```

Výpis 6: Radix-4 iterativní DIT: Matlab kód

2.6.3 Radix-4 iterativní DIT (optimalizace)

```
function y = radix4_iterativni_DIT_optimalizovane(x)
N = length(x);
pocetFazi = log(N)/log(4);
x = digitrevorder(x,4);

for faze=1:pocetFazi
    M = 4^faze;
    pocetSekci = N/M;
    M_ctvrt = M/4;       % hodnotu M/4 si predpocitame
    W = exp(-2i*pi/M); % cast twidle faktoru predpocitame

    for k = 0:M_ctvrt-1; % prehozenim poradi cyklu snizime pocet operaci k vypoctu twidle faktoru
        % spocitame twidle faktory pro jednotlive cleny spolecne pro vsechy sekce
        W1 = W^(k);
        W2 = W^(2*k);
        W3 = W^(3*k);

        for sekce = 0:pocetSekci-1;
            % indexy prvku, ze kterych budeme pocitat motylky (4 bodove DFT)
            n = (sekce*M) + 1 + k + M_ctvrt.*(0:3);

            % hodnoty pronasobime twiddle faktorem W = e^(-i*2pi*k/M) s patricnym posunem
            x0 = x(n(1));
            x1 = x(n(2)) * W1;
            x2 = x(n(3)) * W2;
            x3 = x(n(4)) * W3;

            % opakovane soucty si predpocitame, viz publikace
            % Heckbert, Paul "Fourier Transforms and the Fast Fourier Transform (FFT) Algorithm,"
            x0_plus_x2    =      x0 + x2;
            x0_minus_x2   =      x0 - x2;
            x1_plus_x3    =      x1 + x3;
            i_x1_minus_x3 = 1i*( x1 - x3);

            % souctovy vzorec z predpocitanych hodnot
            x(n(1)) = x0_plus_x2 +  x1_plus_x3;
            x(n(2)) = x0_minus_x2 - i_x1_minus_x3;
            x(n(3)) = x0_plus_x2 -  x1_plus_x3;
            x(n(4)) = x0_minus_x2 + i_x1_minus_x3;

        end;
    end;
end;
y=x;
```

Výpis 7: Radix-4 iterativní DIT (optimalizovaný): Matlab kód

2.6.4 Radix-4 iterativní DIF

Narozdíl od DIT postupujeme obráceně, vektor v každé fázi dělíme na větší počet menších sekcí, ve kterých vypočítáváme jednotivé motýlky, viz 14. Výpočet mění pořadí prvků, narozdíl od DIT provádíme bitové přeuspořádání až na závěr.

```
function y = radix4_iterativni_DIF(x)
N = length(x);
pocetFazi = log(N)/log(4);
M = N;                          % na pocatku bude pocet prvku na sekci M shodny s N

for faze=1:pocetFazi
    pocetSekci = N/M;           % V kazde fazi se pocet sekci zvysi
    W = exp(-2i*pi/M);          % Cast twiddle factoru muzeme predpocitat

    for sekce = 0:pocetSekci-1; % pro kazdou sekci spocitame jednotlive motylky
        for k = 0:M/4-1;        % posun uhlu twidle faktoru

            % vypocteme indexy prvku, ze kterych budeme pocitat motylka
            n = sekce*M + 1 + k + M/4 .* (0:3);
            x0 = x(n(1));       % pro prehlednost si hodnoty ulozime lokalne
            x1 = x(n(2));
            x2 = x(n(3));
            x3 = x(n(4));

            % aplikujeme souctovy vzorec
            a0 = x0 +   x1 + x2 +   x3;
            a1 = x0 - 1i*x1 - x2 + 1i*x3;
            a2 = x0 -   x1 + x2 -   x3;
            a3 = x0 + 1i*x1 - x2 - 1i*x3;

            % pronasobime Wnk a ulozime do puvodniho pole
            x(n(1)) = a0;
            x(n(2)) = a1*W^(k);
            x(n(3)) = a2*W^(2*k);
            x(n(4)) = a3*W^(3*k);

        end;
    end;
    M = M/4; % v kazde fazi snizime pocet prvku o ctvrtinu (odpovida vetveni stromu)
end;
y=digitrevorder(x,4); % na zaver provedeme pretrideni prvku pomoci bit-reversal
```

Výpis 8: Radix-4 iterativní DIF: Matlab kód

2.7 Split-Radix (SRFFT)

Tento algoritmus je založen na pozorování, že v každé fázi výpočtu má Radix-4 lepší výsledky pro liché DFT koeficienty a Radix-2 je lepší pro sudé DFT koeficienty.

Celý problém lze rozdělit na tři subproblémy, které se rekurzivně řeší dle níže uvedených vzorců. Sudé prvky spočítáme pomocí Radix-2 motýlka. Liché následně pomocí Radix-4, kdy na výstupu všechny hodnoty zkombinujeme jako čtyřbodové DFT. Více zde [2], [1].

Algoritmus je minimálně o 20% rychlejší než Radix-2 (při vyšších řádech až o 40%) a jeho složitost je $4N \log N$.

$$X(2k) = \sum_{n=0}^{\frac{N}{2}-1} \left[(n) + x\left(n + \frac{N}{2}\right) \right] .w^{4kn}$$

$$X(4k+3) = \sum_{n=0}^{\frac{N}{4}-1} \left[g(n) + if(n) \right] .w^{3n} .w^{4kn}$$

$$X(4k+1) = \sum_{n=0}^{\frac{N}{4}-1} \left[g(n) - if(n) \right] .w^{n} .w^{4kn}$$

$$\forall f, g :$$

$$g(n) = x(n) - x\left(n + \frac{N}{2}\right)$$

$$f(n) = x\left(n + \frac{N}{4}\right) - x\left(n + \frac{3N}{4}\right)$$

2.8 Split-Radix (SRFFT) - rekurzivní

```
function y= split_radix_rekurzivni (x)
% Split-Radix FFT, rekurzivni

% velikost vektoru v rekurzivnim vetvi
N = length(x);

if N == 1
    y = x;
elseif N==2
    x0 = x(1);
    x1 = x(2);

    y=[(x0 + x1), (x0 - x1)];
else

    % vstup rozdelime na tri posloupnosti, N/2 ze sudych prvku a dve N/4 z lichych prvku.
    sude = x(1:2:(N-1));   % sude prvky: x[0], x[2], x[4], ..., x[N-1].
    liche1 = x(2:4:(N-2)); % liche prvky : x[1], x[5], x[9], ..., x[N-2].
    liche3 = x(4:4:N);     % liche prvky : x[3], x[7], x[11], ..., x[N].

    % rekurzivne volame transformaci
    transformovaneSude = split_radix_rekurzivni(sude);
    transformovaneLiche1 = split_radix_rekurzivni(liche1);
    transformovaneLiche3 = split_radix_rekurzivni(liche3);

    % predpocitame cast twidle faktoru pro vsechny k
    k = 0:(N/4)-1;
    W = exp(-2i*pi*k/N);

    % liche prvky pronasobime twidle faktorem
    transformovaneLiche1 = transformovaneLiche1 .* W;
    transformovaneLiche3 = transformovaneLiche3 .* W.^3;

    % provedeme soucty nad lichymi hodnotami
    transformovaneLiche=[transformovaneLiche1 + transformovaneLiche3, 1i*(transformovaneLiche3 -
        transformovaneLiche1)];

    % a secteme s lichymi
    y = [(transformovaneSude + transformovaneLiche), (transformovaneSude - transformovaneLiche)];
end
```

Výpis 9: Split-Radix rekurzivní: Matlab kód

2.9 Split-Radix (SRFFT) - iterativní

```
function y = split_radix_iterativni_DIF (x)
% Split-Radix DIF FFT, iterativni verze

N = length(x);
pocetFazi = log(N)/log(4);

% na pocatku bude pocet prvku na sekci shodny s N
M = N;

for faze=1:pocetFazi

    pocetSekci = N/M;

    % pro kazdou sekci spocitame jednotlive motylky
    for sekce = 0:pocetSekci-1;

        % posun uhlu twidle faktoru
        for k = 0:M/4-1;

            % vypocteme indexy prvku, ze kterych budeme pocitat motylka
            % napr. 1. motylek = x[0] x[4] x[8] x[12] (v Matlabu indexy [1 5 9 13])
            n = (sekce*M) + 1 + k + M/4.*(0:3);

            % hodnoty, ze kterych budeme pocitat DFT
            x0 = x(n(1));
            x1 = x(n(2));
            x2 = x(n(3));
            x3 = x(n(4));

            % aplikujeme vzorec pro vypocet motylka
            x(n(1)) = x0 +    x1 + x2 +    x3;
            x(n(2)) = (x0 - 1i*x1 - x2 + 1i*x3) * exp(-2i*pi*k/M);
            x(n(3)) = x0 -    x1 + x2 -    x3;
            x(n(4)) = (x0 + 1i*x1 - x2 - 1i*x3) * exp(-2i*pi*3*k/M);
        end;
    end;

    % v kazde fazi snizime pocet prvku na ctvrtinu
    M = M/4;
end;

% na zaver predtridime pole pomoci bit-reverse
y=digitrevorder(x,4);
```

Výpis 10: Split-Radix iterativní: Matlab kód

2.10 Fast Hartley (FHT)

V r. 1942 vyvinul svou alternativu blízkou FFT R. V. L. Hartley a r. 1983 R. N. Bracewell postavil na těchto základech diskrétní verzi tohoto algoritmu.

Zatímco dříve uvedené algoritmy pracovaly s komplexními čísly, FHT upravuje výpočet tak, že si vystačí s polem reálných čísel. Zbylé principy jsou pak obdobné jako u Radix algoritmů.

FHT se tedy liší v některých zásadních principech oproti dříve uvedeným algoritmům:

- Transformaci počítá *z reálných čísel* oproti číslům komplexním, výpočet tedy může být paměťově méně náročný,
- *přímá a inverzní transformace je identická* oproti DFT a IDFT, které jsou různé,
- *výpočetní složitost je nižší*, protože se nemusí zabývat imaginárními složkami,
- pro přenos dat mohou být *na obou stranách stejné HW zařízení*, narozdíl od dříve uvedených, kde je HW vysílač a přijímač různý.

2.11 Další algoritmy

2.11.1 Quick Fourier Transformation (QFT)

Zatímco dříve uvedené algoritmy byly založeny na periodických vlastnostech funkcí $\sin / \cos$, QFT vychází ze symetrie těchto funkcí:

$$\cos\left(\frac{2\pi(N-n)k}{N}\right) = \cos\left(\frac{2\pi nk}{N}\right),$$
$$\sin\left(\frac{2\pi(N-n)k}{N}\right) = -\sin\left(\frac{2\pi nk}{N}\right).$$

2.11.2 Bluestein

Bluestein umí počítat FFT pro libovolný počet prvků, včetně prvočísel. Je založen na konvoluci.

3 Přehled FFT knihoven

Úvodem zmíníme možnosti paralelizace, jež následující knihovny využívají.

OpenMP je multiplatformní API pro vývoj paralelních aplikací se sdílenou pamětí. Paralelní bloky se v kódu definují deklarativně pomocí *#pragma direktiv*, o samotnou tvorbu vláken, rozdělení úloh a jejich synchronizaci se stará API.

Vlákna (POSIX) jsou standardem pro vývoj paralelních aplikací se sdílenou pamětí. Oproti OpenMP se o tvorbu vláken, jejich úloze, synchronizaci a ukončení stará programátor.

MPI (Message Passing Interface) představuje standard pro vývoj aplikací s distribuovanou pamětí. Jednotlivé instance mezi sebou komunikují systémem zasílání zpráv.

Hybridní MPI + OpenMP používá většina knihoven pro 2D a vyšší FFT transformace, přičemž využívá MPI rozhraní pro dekompozici 2D FFT na jednotlivé 1D FFT a jejich distribuci na výpočetní uzly. Pro lokální 1D FFT pak používá OpenMP, neboť má mnohem menší režii pro synchronizaci úloh, menší 1D FFT transformace pomocí MPI by mohla být díky této režii pomalejší než sekvenční výpočet.

3.1 FFTW 3.x

Nejznámější open-source knihovna FFTW (Fastest Fourier Transform in the West). Její jednorozměrnou FFT transformaci využívá řada paralelních knihoven zaměřených na 2D a 3D transformace. Komerčně je tato knihovna využívána např. v programu Matlab, kde řeší všechny FFT operace. Vzhledem k popularitě této knihovny nabízejí významní výrobci hardware (Intel, AMD, IBM) ve svých knihovnách vlastní rozhraní, implementující část funkcí FFTW tak, aby šly programy napsané pro FFTW přeložit s jejich knihovnami a získat tím optimalizace pro danou platformu.

- Licence:
  - Open-source GNU GPL,
  - Komerční MIT.
- FFT: 1D, 2D, 3D
- Hardware: CPU s optimalizacemi SSE/SSE2/Altivec, AVX, ARM Neon
- Způsob výpočtu:
  - sekvenční,
  - paralelní se sdílenou pamětí (OpenMP, vlákna),
  - paralelní s distribuovanou pamětí (MPI),
  - hybridní MPI + OpenMP.
- Omezení na délku vstupu: žádné, vstupem mohou být libovolně velká čísla včetně prvočísel

- Jazyk:
  - C, C++, Fortran
  - Adaptéry: C#, Java, Python, JavaScript, Perl, Ruby, PHP, ADA a další
- Závislost na jiné FFT knihovně: ne
- Web: `http://www.fftw.org`

3.2 MKL - Intel Math Kernel Library

Komerční knihovna, dostupná na obou Ostravských superpočítačích Anselm i Salomon.

- Licence: Komerční
- FFT: 1D, 2D, 3D
- Hardware: CPU + všechny Intel optimalizace včetně přímé podpory MIC akcelerátoru (Intel Phi)
- Způsob výpočtu:
  - sekvenční,
  - paralelní se sdílenou pamětí (OpenMP, vlákna),
  - paralelní s distribuovanou pamětí (MPI),
  - hybridní MPI + OpenMP.
- Jazyk:
  - C, C++, Fortran
  - Adaptéry: C#, Java, Python
- Závislost na jiné FFT knihovně: ne
- Kompatibilita: nabízí kompatibilní rozhraní s FFTW. Aplikaci můžeme vyvinout pomocí volně dostupné FFTW knihovny na vlastním počítači a tento kód zkompilovat na superpočítači pomocí Intel MKL knihoven a hlavičkových souborů, čímž získáme vysoce optimalizované řešení pro prostředí superpočítače včetně podpory MIC akcelerátorů.
- Web: `https://software.intel.com/en-us/intel-mkl`

3.3 AccFFT

Nová masivně paralelní knihovna pro CPU i GPU.

- Licence: Open-source GNU GPL
- FFT: 1D, 2D, 3D
- Hardware: CPU, GPU
- Způsob výpočtu:
  - sekvenční,
  - paralelní se sdílenou pamětí (OpenMP, vlákna),

  - paralelní s distribuovanou pamětí (MPI),
  - hybridní MPI + OpenMP,
  - grafický akcelerátor.
- Jazyk: C/C++
- Závislost na jiné FFT knihovně: FFTW (pro 1D FFT)
- Web: `http://accfft.org/`

3.4 GPUFFTW

Knihovna využívající NVIDIA GPU namísto klasických CPU, dle výsledků zveřejněných na webu autorů dosahují výrazně vyššího výkonu než FFTW i MKL pro 1D transformaci.

- Licence: zdarma pro nekomerční využití, licence GLUT
- FFT: 1D
- Hardware: GPU
- Způsob výpočtu: sekvenční/paralelní přes grafický akcelerátor (sdílená paměť)
- Jazyk: C/C++ (CUDA)
- Závislost na jiné FFT knihovně: ne
- Web: `http://gamma.cs.unc.edu/GPUFFTW/`

3.5 FFTE

Japonská verze FFTW (Fastest Fourier Transform in the East).

- Licence:
  - Open-source,
  - zdarma i pro komerční využití.
- FFT: 1D, 2D, 3D
- Hardware: CPU
- Způsob výpočtu:
  - sekvenční,
  - paralelní se sdílenou pamětí (OpenMP, vlákna),
  - paralelní s distribuovanou pamětí (MPI),
  - hybridní MPI + OpenMP.
- Jazyk: FORTRAN
- Závislost na jiné FFT knihovně: ne
- Web: `http://www.ffte.jp/`

3.6 ACML – AMD Core Math Library

Komerční matematická knihovna od AMD.

- Licence: Komerční
- FFT: 1D, 2D, 3D
- Hardware: CPU – optimalizace pro AMD
- Způsob výpočtu:
  - sekvenční,
  - paralelní se sdílenou pamětí (OpenMP, vlákna),
  - paralelní s distribuovanou pamětí (MPI),
  - hybridní MPI + OpenMP.
- Jazyk: FORTRAN
- Závislost na jiné FFT knihovně: ne
- Kompatibilita: podobně jako Intel MKL nabízí rozhraní pro FFTW, díky kterému je možné bez úprav přejít z FFTW na tuto knihovnu
- Web: `http://developer.amd.com/tools-and-sdks/archive/amd-core-math-library-acml/`

3.7 2DECOMP&FFT

Knihovna zaměřená na dekompozici 2D a 3D FFT pomocí MPI.

- Licence:
  - Open-source,
  - zdarma i pro komerční využití.
- FFT: 1D, 2D, 3D
- Hardware: CPU
- Způsob výpočtu:
  - sekvenční,
  - paralelní se sdílenou pamětí (OpenMP),
  - paralelní s distribuovanou pamětí (MPI),
  - hybridní MPI + OpenMP.
- Jazyk: FORTRAN
- Závislost na jiné FFT knihovně: knihovna má vlastní základní 1D FFT, přesto doporučuje pro 1D FFT využít některou z optimalizovaných knihoven třetích stran (FFTW, MKL, ACML, ESSL, FFTPACK, FFTE aj.)
- Web: `http://www.2decomp.org`

3.8 P3DFFT

Knihovna zaměřená na škálovatelné 2D a 3D FFT transformace pomocí MPI.

- Licence: Open-source
- FFT: 1D, 2D, 3D
- Hardware: CPU
- Způsob výpočtu:
  - sekvenční,
  - paralelní se sdílenou pamětí (OpenMP, vlákna),
  - paralelní s distribuovanou pamětí (MPI),
  - hybridní MPI + OpenMP.
- Jazyk: C, C++
- Závislost na jiné FFT knihovně: FFTW nebo ESSL pro 1D FFT transformace
- Web: `http://www.p3dfft.net`

3.9 Kiss FFT

Jednoduchá knihovna nabízející různé implementace FFT algoritmů.

- Licence:
  - Open-source BSD,
  - Komerční.
- FFT: 1D
- Hardware: CPU
- Způsob výpočtu:
  - sekvenční,
  - paralelní se sdílenou pamětí.
- Jazyk: C, C++
- Závislost na jiné FFT knihovně: ne
- Web: `https://sourceforge.net/projects/kissfft/`

3.10 OpenFFT

Knihovna zaměřená na 3D a 4D paralelní FFT pomocí MPI.

- Licence: Open-source
- FFT: 3D, 4D
- Hardware: CPU
- Způsob výpočtu:

  - paralelní s distribuovanou pamětí (MPI)
  - hybridní MPI + OpenMP
- Jazyk: C, C++, FORTRAN
- Závislost na jiné FFT knihovně: ne
- Web: `http://www.openmx-square.org/openfft/`

3.11 IBM ESSL

Komerční IBM knihovna dostupná pouze na IBM hardware.

- Licence: Komerční
- FFT: 1D, 2D, 3D
- Hardware: CPU
- Způsob výpočtu:
  - sekvenční,
  - paralelní se sdílenou pamětí (OpenMP, vlákna),
  - paralelní s distribuovanou pamětí (MPI),
  - hybridní MPI + OpenMP.
- Jazyk: C, C++, FORTRAN
- Závislost na jiné FFT knihovně: ne
- Kompatibilita: FFTW adaptér, podobně jako Intel MKL a AMD ACML nabízí možnost kompilace kódu napsaného pro FFTW s využitím této knihovny
- Web: `http://www-03.ibm.com/systems/power/software/essl/`

3.12 FFTPACK

Starší knihovna, kterou je doporučeno využívat pouze z kompatibilních důvodů se starším kódem.

- Licence: Public domain
- FFT: 1D, 2D
- Hardware: CPU
- Způsob výpočtu:
  - sekvenční,
  - paralelní se sdílenou pamětí.
- Jazyk:
  - FORTRAN
  - Adaptéry: C, Java
- Závislost na jiné FFT knihovně: ne

- Web: `http://www.netlib.org/fftpack/`

3.13 JTransforms

Paralelní knihovna kompletně implementovaná v Javě.

- Licence: Public domain
- FFT: 1D, 2D, 3D
- Hardware: CPU
- Způsob výpočtu:
  - sekvenční,
  - paralelní se sdílenou pamětí.
- Jazyk: Java
- Závislost na jiné FFT knihovně: ne
- Web: `https://sites.google.com/site/piotrwendykier/software/jtransforms`

4 Problematika násobení obrovských čísel

Násobení obrovských čísel se objevuje v mnoha odvětvích, jedná se zejména o šifrování, teorii čísel a další. Tradiční přístupy požadují N^2 operací, kde N je počet číslic. Uveďme si příklad:

$$\begin{array}{r} 1234 \\ \times 2125 \\ \hline 5 \times 1000 + 5 \times 200 + 5 \times 30 + 5 \times 4 \\ +20 \times 1000 + 20 \times 200 + 20 \times 30 + 20 \times 4 \\ +100 \times 1000 + 100 \times 200 + 100 \times 30 + 100 \times 4 \\ +2000 \times 1000 + 2000 \times 200 + 2000 \times 30 + 2000 \times 4 \\ \hline 2622250 \end{array} \qquad \begin{array}{r} \text{počet operací} \\ \text{násobení je:} \end{array} \qquad \begin{array}{r} 4 \\ \times 4 \\ \hline 16 \end{array}$$

Násobení dvou celých čísel představuje časovou složitost $O(N^2)$, kdy doba trvání průběhu se zvyšuje kvadraticky, tj. složitost, která se v případě obrovských čísel hodně rychle zvětšuje, a proto hledáme výkonnější algoritmy, které by nám umožnily dosáhnout součinu efektivnějším způsobem. Jednou z možností je právě použití FFT algoritmů se složitostí $O(N \log N)$.

Tady se ještě na chvíli zastavíme a podíváme se, jak to vlastně s tím klasickým násobením je. Nejprve ale nahlédneme do minulosti.

4.1 Historie matematiky a násobení

Dějiny matematiky se nedatují pouze od prvních pokusů pravěkého člověka spočítat úlovek, přes velký vzestup matematiky ve Starém Řecku až ke dnešním, moderním matematickým oborům zaměstnávajících obrovský počet matematiků, ale je zřejmé, že matematika není výlučnou doménou pouze lidského druhu, neboť bylo rovněž dokázáno, že vrány umějí rozlišovat mezi množinami až o čtyřech prvcích, tudíž základy počítání ovládají i jiní tvorové ([3]).

Zatímco v pravěku šlo o vědomé vyjadřování počtu dvou, tří, později čtyř až pěti kusů předmětů, a všechno ostatní už znamenalo neurčitě mnoho, ve starověku vznikaly aritmetické pojmy, kvantitativní geometrické vztahy a operace s nimi.

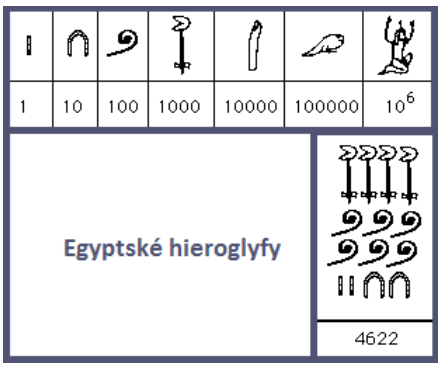
4.1.1 Mezopotámie

První písemné památky v historii matematiky pocházejí z Mezopotámie. Z období 2200 až 1800 př. n. l. se dochovalo velké množství matematických tabulek, které svědčí o pokročilé algebře a

geometrie. K násobení používali důmyslné komplety tabulek a dělení se převádělo na násobení převrácenou hodnotou, k čemuž sloužily opět tabulky. Pracovalo se s desítkovou a šedesátkovou soustavou. Objevy naznačují, že už tenkrát se odborníci zabývali problémy Pythagorovy věty ([3], [16]).

4.1.2 Egypt

Dochovaný papyrus nám umožňuje nahlédnout do historie, kdy Egypťané používali desítkovou číselnou soustavu, pracovali s mocninami desíti od $10^0 - 10^6$, přičemž byly používány **hieroglyfy**, tj. speciální znaky, např. jedničku zastupuje hieroglyf měřící hole, desítku pak znak kravích pout, atd. Na obrázku najdeme jako příklad číslo 4622 ([3], [17]).



Obrázek 15: Egyptské číselné hieroglyfy

Násobení pak prováděli tak, že např. u součinu 24×35 si připravili dva sloupce, v pravém z nich je větší číslo, které vždy na dalším řádku zdvojnásobíme, zatímco v levém sloupci začínáme od jedničky a postupujeme stejně (každý následující řádek je násobkem dvou). S násobením skončíme, jakmile zjistíme, že v levém sloupci jsme již schopni najít a sečíst postupně čísla tak, aby součet byl roven číslu 24 (tj. binární zápis čísla 24) a tyto řádky si označíme. Výsledek násobení 24×35 bude součet označených řádků ([18]).

24	$\times$ 35
1	35
2	70
4	140
8	**280**
16	**560**
Výsledek	**840**

Tabulka 5: Egyptská násobilka

4.1.3 Indie

Indická matematika byla velmi rozvinutá, došlo k vytvoření pozičního systému, znalosti prvních devíti číslic a obrovským objevem indických matematiků se stala nula.

4.1.4 Řecko

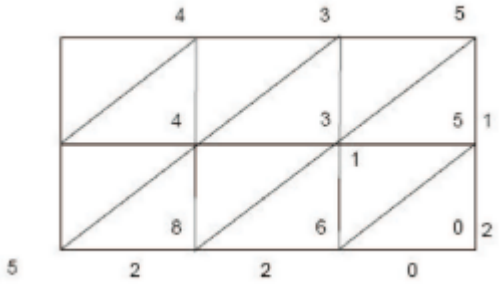
Starověké Řecko je považováno za kolébku evropské kultury a vzdělanosti. Nejproslulejší v oblasti matematiky jsou *Eukleidovy Základy*, jedná se o soubor třinácti knih, které se staly na více než dva tisíce let směrodatným textem o geometrii a říkalo se o něm, že je to nejúspěšnější učebnice v historii matematiky. Další významnou osobností byl *Pythagoras ze Samu*, přezdívaný otec čísel, nebo nebo např. *Archimédes.*

Postupně vznikají různé metody násobení, které se šíří i do okolních zemí a používají se dodnes. Pro zajímavost si uvedeme příklad **čínské** a **indické násobilky.**

Indická násobilka

Chtějme vynásobit čísla 435×12. Vytvoříme si obdélník s buňkami pro násobení jednotlivých cifer a každou buňku rozpůlíme na dva trojúhelníky. Při násobení každé číslice s každou pak zapisujeme průběžné výsledky do spodního trojúhelníku patřičné buňky v případě jednociferného výsledku a v případě dvojciferného součinu pak vyšší řád umístíme do horního trojúhelníku.

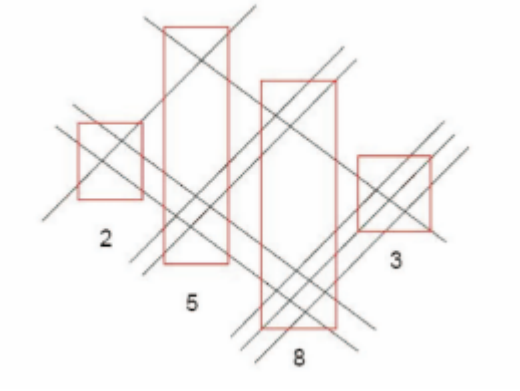
Následně sečteme číslice pod úhlopříčkou. Výsledek je 5220.



Obrázek 16: Příklad indického násobení, rozšířeného také v dalších zemích

Čínská násobilka

Nechť požadujeme součin čísel 123×21. Vedeme jednu rovnoběžku v řádu stovek, o kus dál dvě rovnoběžky v řádu desítek a tři rovnoběžky za jednotky (tj. číslo 123). Kolmo na ně pak vyjádříme číslo 21 jako dvě rovnoběžky charakterizující desítky a jednu rovnoběžku jako jednotku. Spočteme průsečíky ve vyznačených množinách [5] a získáme výsledek je 2583. (Více zde [20].)



Obrázek 17: Příklad čínského násobení

4.2 Klasické násobení

Vrátíme se na začátek čtvrté kapitoly. Vysvětlili jsme si, že použití FFT algoritmů pro násobení obrovských čísel je jednou z možností, jak dosáhnout součinu efektivnějším způsobem se složitostí $O(N \log N)$ oproti původní $O(N^2)$, s níž klasické násobení probíhá. Rovněž jsme si uvedli příklad, včetně demonstrace postupu výpočtu, jako názornou ukázku.

Úplně nejjednodušeji vynásobíme dvě čísla tak, že zadáme $a.b$ a procesor se svými instrukcemi vrátí výsledek, protože to prostě už umí, dokáže součin vypočítat velmi rychle, neboť postupem času bylo dosaženo vysoce efektivního algoritmu. Tento způsob výpočtu je rychlý. Ale omezený. A omezená je právě velikost N, to znamená, že záleží na velikosti čísel, které potřebujeme vynásobit. Pro hodně velká N již tento způsob násobení použít nelze, a to právě díky tomu, že čísla se již nevejdou do paměti. Je třeba najít jiné možnosti, jak provést násobení obrovských čísel, nebo dokonce jak je násobit co nejúsporněji.

[5] Pokud by byl součet průsečíků v některé množině větší než 9, zapsali bychom jednotky a cifru označující desítky bychom převedli k následující množině umístěné nalevo.

Ukázkový příklad z úvodu této kapitoly znázorňuje násobení dvou celých čísel se složitostí $O(N^2)$, které lze ilustrovat pomocí následujícího kódu.

```
function y = klasicke_nasobeni_mensi_cisla(a, b)
% pronasobi dve cisla po slozkach tak, jak se provadi u klasickeho nasobeni

suma = 0;
for j=1:length(b)
    for i=1:length(a)
        suma = suma + a(i)*10^(j-1) * b(j)*10^(i-1);
    end
end

y = suma;
```

Výpis 11: Klasické násobení pro menší čísla: Matlab kód

Pro extrémně velká čísla si však s tímto algoritmem nevystačíme a je třeba číslo rozložit na jednotlivé koeficienty a využít vektorového zápisu, což vidíme v následujícím algoritmu:

```
function y = nasobeni_pomoci_konvoluce_vektoru(a, b)
% nasobeni dvou velkych cisel reprezentovanych polynomy pomoci konvoluce vektoru

% vytvorime novy vektor, do ktereho budeme ukladat vysledky
vysledek = zeros(1, length(a) + length(b));

% nejprve provedeme klasickou konvoluci dvou vektoru, pozn. Matlab indexuje od 1
for j = 1:length(b)
    for i = 1:length(a)
        % nasobky prubezne pricitame k vysledkum z predchozi iterace
        rad = i+j-1;
        vysledek(rad) = a(i) * b(j) + vysledek(rad);
    end
end

% pote secteme slozky tak, aby se zachovala logika presunu do vyssiho radu
% napr. 15 -> 5 zustane v radu n0, 1 se pricte k radu n1
y = soucet_slozek_velkeho_cisla(vysledek);
```

Výpis 12: Násobení velkých čísel pomocí vektorové konvoluce: Matlab kód

```
function y = soucet_slozek_velkeho_cisla(x)

N = length(x);
vysledek = zeros(1, N+1);
presunDoVyssihoRadu = 0;

for i = 1:N

    % k soucasnemu cislu pricteme cislo presunute z nizsiho radu
    soucet = x(i) + presunDoVyssihoRadu;

    % cislo daneho radu ulozime do vysledneho pole napr.
    % soucet = 15 => 5 zapiseme na pozici radu pro dilci nasobeni
    vysledek(i) = mod(soucet, 10);

    % cislo, ktere presahuje zaklad presuneme do vyssiho radu, napr. soucet = 15 => 1 jde dale
    presunDoVyssihoRadu = floor(soucet / 10);

end

vysledek(N+1) = presunDoVyssihoRadu;

y = vysledek;
```

Výpis 13: Násobení velkých čísel pomocí vektorové konvoluce využívá funkci součtu jednotlivých složek velkého čísla: Matlab kód

Ani s tímto postupem se však nespokojíme, neboť sice funguje pro extrémně velká čísla, nicméně doba výpočtu se kvadraticky zvyšuje, což právě v případě těchto extrémně velkých dat je velmi nežádoucí, a právě toto je důvod, proč se zabýváme možnostmi užití FFT algoritmů.

Vysvětlili jsme si, jak tato metoda funguje a pokusíme se zjistit, pro jak velká N bude FFT násobení rychlejší než klasické násobení a kolik času ušetří.

Budeme provádět experimenty v závislosti na použité implementaci FFT (a následně také inverzní FFT), zaměříme se na porovnávání velikosti vstupu (délky vektoru) a základu, který si budeme vybírat pro reprezentaci celého čísla, tzn. jak který parametr ovlivňuje dobu běhu programu násobícího algoritmu.

4.3 Základní pojmy

V této části si popíšeme metodu celočíselného násobení pomocí FFT:

- Nejprve si ukážeme, jak FFT násobení funguje pro polynomy.

- Dále budeme reprezentovat celé číslo jako polynom.
- A nakonec si celý postup FFT násobení ukážeme na názorných příkladech o různém základu.

Začneme definováním některých základních pojmů týkajících se polynomů.

4.3.1 Polynomy

Definice 4.1 *Polynom, nebo také mnohočlen, je výraz ve tvaru*

$$a(x) = \sum_{i=0}^{N-1} a_i x^i = a_0 + a_1 x + a_2 x^2 + ... + a_{N-1} x^{N-1},$$

kde x je základ polynomu a čísla $a_0, a_1, ..., a_{N-1}$ se nazývají koeficienty polynomu.

Polynomy jsou obvykle reprezentovány pomocí *vektoru koeficientů* $a = [a_0, a_1, ..., a_{N-1}]$ a budeme je indexovat od 0 po $N-1$ u polynomu o délce N členů, což odpovídá velikosti polynomu $a(x)$. Můžeme říci, že hodnota polynomu $a(x)$ má časovou složitost $O(N)$, protože potřebujeme vyhodnotit N operací násobení.

Stupněm takového polynomu *deg* $a(x)$ je hodnota i, kde $a_i \neq 0$ a platí:

$$\forall j > i, a_j = 0.$$

Jedná se o největší index nenulového koeficientu a_i. Můžeme také říci, že stupeň polynomu je exponent největší mocniny, která se v polynomu objevila.

Vynásobením dvou polynomů získáme třetí polynom. Tento proces se nazývá vektorová konvoluce a stejně tak jako násobení celých čísel, vektorová konvoluce má složitost $O(N^2)$.

Definice 4.2 *Nechť vektor $\boldsymbol{a}$ má N_1 složek $(a_0, a_1, ..., a_{N_1-1})$ a vektor $\boldsymbol{b}$ má N_2 složek $(b_0, b_1, ..., b_{N_2-1})$. Pak vektor $\boldsymbol{c}$ o počtu $N_1 + N_2 - 1$ složek $(c_0, c_1, ..., c_{N_1+N_2-2})$, které vypočteme jako*

$$c_n = \sum_{k=0}^{n} a_k b_{n-k} = \sum_{k=0}^{n} a_{n-k} b_k, n = 0, 1, ..., N_1 + N_2 - 2,$$

nazveme konvolucí vektorů $\boldsymbol{a}$ a $\boldsymbol{b}$.

Princip spočívá v tom, že pro každou mocninu základu x určíme kombinaci indexů obou vektorů (kombinaci jednotlivých členů z obou mnohočlenů), které násobíme každý prvek s každým. Každá kombinace je multiplikativní operací, kterou je nutné započítat.

Dále víme, že počet kombinací polynomu stupně (N-1), odpovídající nejvyšší mocnině x vyskytující se v polynomu, vykazuje asymptotickou složitost $O(N)$, neboť dle definice asymptotické složitosti platí, že zanedbáváme multiplikativní i aditivní konstanty, tzn. $O(N + 1000) = O(N.1000) = O(N)$.

Násobení dvou polynomů o N členech pak představuje (N-1)-tý stupeň každého z polynomů, vzniká N^2 multiplikativních operací a ($2N$-2)-tý stupeň ($deg(c)$) výsledného polynomu $c(x)$ o ($2N$-1) členech:

$$c(x) = a(x)b(x) = a_0b_0 + (a_0b_1 + a_1b_0)x + (a_0b_2 + a_1b_1 + a_2b_0)x^2 + ... + a_{N-1}b_{N-1}x^{2N-2},$$

neboť součinem dvou členů x^{N-1} bude nejvyšší mocnina výsledného polynomu rovna

$$x^{(N-1)+(N-1)} = x^{2N-2}$$

a víme, že platí:

$$deg(ab) = deg(c) = deg(a) + deg(b).$$

Pro jednoduchost přidáme k výslednému polynomu nakonec nulový člen, čímž získáme $2N$ členů, tj. délka vektoru se bude rovnat $2N$ a stupeň polynomu $2N - 1$.

Násobení dvou polynomů ve formě koeficientů představuje tedy kvadratickou složitost $O(N^2)$. Abychom tuto složitost snížili, je nutné použít jiný způsob reprezentace polynomů a jedním z nich je *Interpolace pomocí polynomu.*

4.3.2 Interpolace pomocí polynomů

Nechť existuje množina N bodů v rovině $S = (x_0, y_0), (x_1, y_1), (x_2, y_2), ..., (x_{N-1}, y_{N-1})$, pro kterou platí, že všechna x_i jsou navzájem různá. Pak existuje jedinečný polynom $p(x)$ stupně $N - 1$, kde $p(x_i) = y_i$, pro $i = 0, 1, ..., N - 1$.

Interpolační teorém říká, že máme -li $(N - 1)$-tý stupeň polynomu a vyhodnotíme ho v N různých bodech, pak všech N vstupů a výstupů představuje unikátní reprezentaci tohoto polynomu.

Víme, že součinem polynomů p a q stupňů $N-1$ je polynom stupně $2N-1$. Dále víme, že tento výsledný polynom je jednoznačně definován pomocí navzájem různých $2N$ bodů. Čímž získáváme základní představu o FFT násobení:

- Vyhodnotíme p ve všech $2N$ bodech
- Vyhodnotíme q ve všech $2N$ bodech
- Spočítáme $2N$ součinů všech p a q hodnot získaných z $2N$ bodů.

Stále ještě nejsme u konce, neboť hodnocení $2N$ různých vstupů bude znamenat stále časovou složitost $O(N^2)$. Našim cílem je tedy najít množinu, která má specifické vlastnosti, takže můžeme opětovně použít některé výstupy k vyhodnocení jiných částí vstupu. Specifickou množinou vstupních hodnot použitých při FFT jsou *primitivní kořeny jedničky.*

4.3.3 Primitivní odmocniny z jedničky

Definice 4.3 *Řekneme, že číslo ω je primitivní N-tá odmocnina z 1 [12], [15], kde $N \geq 2$, platí-li:*

- $\omega^N = 1$*, tzn. jedná se o N-tou odmocninu z 1*
- *žádné z čísel $\omega^1, \omega^2, ..., \omega^{N-1}$ není rovno 1*

Pohybujeme se v komplexní rovině, kde komplexní číslo z je číslo ve tvaru

$$z = x + iy, \text{kde } x, y \in \mathbb{R} \text{ a } i^2 = -1.$$

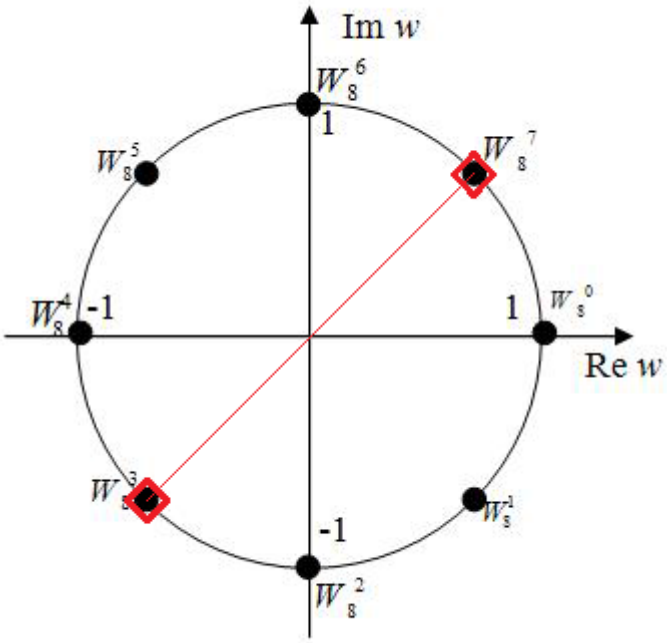
Číslo x nazýváme reálnou a číslo y imaginární částí komplexního čísla z a značíme Re z a Im z (více zde [13]).

Primitivní N-té odmocniny využijeme při provádění FFT ke snižování počtu hodnot na $N/2$ a zpracovávání pouze první poloviny hodnot, zatímco druhá polovina hodnot bude odvozena z první poloviny pomocí obrácení znaménka, neboť je-li ω N-tým kořenem a $N \geq 2$, platí:

$$\omega^{k+N/2} = -\omega^k.$$

Je-li N sudé číslo, pak polynom $a(x)$ o délce N

$$a(x) = \sum_{i=0}^{N-1} a_i x^i = a_0 + a_1 x + a_2 x^2 + ... + a_{N-1} x^{N-1}$$



Obrázek 18: N-tá odmocnina z 1 pro $N = 8$([14])

rozdělíme na dva polynomy, kde první závorka bude obsahovat členy se sudými exponenty, zatímco druhá závorka ty s lichými:

$$a(x) = (a_0x^0 + a_2x^2 + ... + a_{N-2}x^{N-2}) + (a_1x^1 + a_3x^3 + ... + a_{N-1}x^{N-1}).$$

Po vytknutí x před druhou závorku získáme v obou závorkách pouze sudé mocniny:

$$a(x) = (a_0x^0 + a_2x^2 + ... + a_{N-2}x^{N-2}) + x(a_1x^0 + a_3x^2 + ... + a_{N-1}x^{N-2}),$$

tzn. můžeme říci, že

$$a(x) = a_s(x^2) + x.a_l(x^2),$$

kde a_s je část polynomu se sudými exponenty a a_l s lichými. A jelikož obě části původního polynomu mají poloviční délku $N/2$, zavedeme $x^2 = t$ a můžeme psát:

$$a_s(t) = (a_0t^0 + a_2t^1 + ... + a_{N-2}t^{\frac{N-2}{2}}),$$

$$a_l(t) = (a_1 t^0 + a_3 t^1 + ... + a_{N-1} t^{\frac{N-2}{2}}).$$

A pro zápornou hodnotu pak použijeme vztah s obráceným znaménkem:

$$a(-x) = a_s(x^2) - x.a_l(x^2).$$

Teď zpátky k našemu ω. Všech N mocnin ω si představíme jako množinu ω^k, kde $k = 0, 1, ..., N-1$ a jedná se o kořeny rovnice $\omega^N = 1$, zmíněné v definici. Komplexní číslo $z = \omega^k = e^{-i\frac{2\pi}{N}k}$ leží v komplexní rovině na kružnici o poloměru 1 ([14]) a jak jsme zmínili v kapitole FFT metody, nazývá se Twiddle faktor a vyjadřuje posunutí po této kružnici.

Pokusme se nyní vyjádřit celé číslo jako polynom a následně provést násobení pomocí FFT.

4.3.4 Reprezentace celého čísla pomocí polynomu

Vrátíme se k definici **polynomu** 4.1. Abychom takový polynom mohli vytvořit, potřebujeme určit **základ** x, který si musíme zvolit, a **koeficienty polynomu** $a_0, a_1, ..., a_{N-1}$.

Uvažujme číslo 1234 a pro jednoduchost si zvolme základ $x = 10$. Označíme si indexy koeficientů [6]

$$\underbrace{1}_{a_3}\ \underbrace{2}_{a_2}\ \underbrace{3}_{a_1}\ \underbrace{4}_{a_0}$$

a zapíšeme koeficienty vektoru $\mathbf{a} = [4, 3, 2, 1]$.

Nyní použijeme stejné číslo a změníme základ na $x = 100$.

$$\underbrace{12}_{b_1}\ \underbrace{34}_{b_0}$$

a zapíšeme koeficienty vektoru $\mathbf{b} = [34, 12]$.

A ještě si ukážeme, jak si poradíme s číslem 12345 pro základ $x = 100$:

$$\underbrace{1}_{c_2}\ \underbrace{23}_{c_1}\ \underbrace{45}_{c_0}$$

[6] Uvědomme si, že číslujeme od jednotek zprava doleva.

a zapíšeme koeficienty vektoru $\mathbf{c} = [45, 23, 1]$ [7].

Obdobně postupujeme u ostatních číselných základů.

4.3.5 Násobení dvou polynomů pomocí FFT

V předchozí části jsme reprezentovali číslo jako polynom. Také již víme, že součinem dvou polynomů stupně $N-1$ je polynom stupně $2N-2$ a po domluvě 4.3.1 budeme pracovat s polynomem stupně $2N-1$ o délce $2N$ členů. Vzhledem k tomu, že součin má vyšší stupeň než jednotliví činitelé, je třeba si pořídit dostatečný počet bodů, čehož docílíme tak, že každý vektor o délce N doplníme nulami na velikost vektoru $2N$:

$$\mathbf{a'} = [\underbrace{a_0, a_1, ..., a_{N-1}, 0, 0, ..., 0}_{2N}], \qquad \mathbf{b'} = [\underbrace{b_0, b_1, ..., b_{N-1}, 0, 0, ..., 0}_{2N}].$$

Nalezneme FFT obrazy **A', B'** vektorů **a'** a **b'**, které násobíme po složkách:

$\mathbf{A'} = \mathrm{FFT}(\mathbf{a'}) = [A_0, A_1, ..., A_{2N-1}]$,
$\mathbf{B'} = \mathrm{FFT}(\mathbf{b'}) = [B_0, B_1, ..., B_{2N-1}]$,

$$\mathbf{A'}.*\mathbf{B'} = [A_0.B_0, A_1.B_1, ..., A_{2N-1}.B_{2N-1}] = [C_0, C_1, ..., C_{2N-1}] = \mathbf{C'}.$$

Zpětnou transformací získáme výsledný vektor **c'** o délce $2N$:

$\mathbf{c'} = \mathrm{IFFT}(\mathbf{C'}) = [c_0, c_1, ..., c_{2N-1}]$

a v závislosti na zvoleném základu dopočteme výsledek, neboť pro součin c rovněž platí, že

$$c(x) = c_0x^0 + c_1x^1 + ... + c_{2N-1}x^{2N-1}.$$

4.3.6 Ilustrace násobení

Jak násobení provedeme?

1) Zvolíme základ x.

2) Číslo a reprezentujeme polynomem

$$a(x) = a_0x^0 + a_1x^1 + ... + a_{N-1}x^{N-1}.$$

[7] Opět číslujeme zprava doleva, takže samotná zůstane cifra charakterizující nejvyšší řád.

Pracovat budeme s vektorem $\mathbf{a} = [\underbrace{a_0, a_1, ..., a_{N-1}}_{N}]$.

Analogicky číslo b vyjádříme pomocí polynomu

$$b(x) = b_0x^0 + b_1x^1 + ... + b_{N-1}x^{N-1}$$

a připravíme si vektor $\mathbf{b} = [\underbrace{b_0, b_1, ..., b_{N-1}}_{N}]$.

Oba vektory doplníme nulami na délku vektoru $2N$:

$$\mathbf{a'} = [\underbrace{a_0, a_1, ..., a_{N-1}, 0, 0, ..., 0}_{2N}], \qquad \mathbf{b'} = [\underbrace{b_0, b_1, ..., b_{N-1}, 0, 0, ..., 0}_{2N}].$$

3) Nalezneme FFT obrazy **A', B'** vektorů **a'** a **b'**:

$\mathbf{A'} = \text{FFT}(\mathbf{a'}) = [A_0, A_1, ..., A_{2N-1}]$,
$\mathbf{B'} = \text{FFT}(\mathbf{b'}) = [B_0, B_1, ..., B_{2N-1}]$.

4) Násobíme obrazy po složkách

$$\mathbf{A'}.*\mathbf{B'} = [A_0.B_0, A_1.B_1, ..., A_{2N-1}.B_{2N-1}] = [C_0, C_1, ..., C_{2N-1}] = \mathbf{C'}.$$

5) Zpětnou transformací získáme výsledný vektor délky $2N$

$\mathbf{c'} = \text{IFFT}(\mathbf{C'}) = [c_0, c_1, ..., c_{2N-1}]$

a v závislosti na zvoleném základu dopočteme výsledek, neboť pro součin c rovněž platí, že

$$c(x) = c_0x^0 + c_1x^1 + ... + c_{2N-1}x^{2N-1}.$$

4.3.6.1 Příklad: Chtějme vynásobit čísla 1234 a 2125.

1) Základ $x = 10$

$$a = \underbrace{1}_{a_3}\underbrace{2}_{a_2}\underbrace{3}_{a_1}\underbrace{4}_{a_0}, \qquad b = \underbrace{2}_{b_3}\underbrace{1}_{b_2}\underbrace{2}_{b_1}\underbrace{5}_{b_0}.$$

2) Číslo $a = 1234$ vyjádříme jako polynom

$$a(x) = a_0x^0 + a_1x^1 + ... + a_{N-1}x^{N-1}.$$

Pracovat budeme s vektorem $\mathbf{a} = [\underbrace{4, 3, 2, 1}_{N=4}]$.

Číslo b vyjádříme pomocí polynomu

$$b(x) = b_0 x^0 + b_1 x^1 + ... + b_{N-1} x^{N-1}$$

a připravíme si vektor $\mathbf{b} = [\underbrace{5, 2, 1, 2}_{N=4}]$.

Oba vektory doplníme nulami na délku vektoru $2N$:

$$\mathbf{a'} = [\underbrace{4, 3, 2, 1, 0, 0, 0, 0}_{2N=8}], \qquad \mathbf{b'} = [\underbrace{5, 2, 1, 2, 0, 0, 0, 0}_{2N=8}]$$

3) Nalezneme FFT obrazy **A', B'** vektorů **a'** a **b'**:

A' = FFT(**a'**) =
$[10.0000 + 0.0000i, 5.4142 - 4.8284i, 2.0000 - 2.0000i, 2.5858 - 0.8284i,$
$2.0000 + 0.0000i, 2.5858 + 0.8284i, 2.0000 + 2.0000i, 5.4142 + 4.8284i]$,

B' = FFT(**b'**) =
$[10.0000 + 0.0000i, 5.0000 - 3.8284i, 4.0000 + 0.0000i, 5.0000 - 1.8284i,$
$2.0000 + 0.0000i, 5.0000 + 1.8284i, 4.0000 + 0.0000i, 5.0000 + 3.8284i]$.

4) Násobíme obrazy po složkách

C' = $100*$
$[1.0000 + 0.0000i, 0.0859 - 0.4487i, 0.0800 - 0.0800i, 0.1141 - 0.0887i,$
$0.0400 + 0.0000i, 0.1141 + 0.0887i, 0.0800 + 0.0800i, 0.0859 + 0.4487i]$.

5) Zpětnou transformací získáme výsledný vektor délky $2N$

$\mathbf{c'} = \text{IFFT}(\mathbf{C'}) = [\underbrace{20}_{c_0}, \underbrace{23}_{c_1}, \underbrace{20}_{c_2}, \underbrace{20}_{c_3}, \underbrace{10}_{c_4}, \underbrace{5}_{c_5}, \underbrace{2}_{c_6}, \underbrace{0}_{c_7}]$

a víme, že základ $x = 10$, tedy

$$c(x) = c_0 x^0 + c_1 x^1 + ... + c_{2N-1} x^{2N-1} =$$

$= 20.10^0 + 23.10^1 + 20.10^2 + 20.10^3 + 10.10^4 + 5.10^5 + 2.10^6 + 0.10^7 = 20 + 230 + 2000 + 20000 + 100000 + 500000 + 2000000 = \underline{2622250}$

4.3.6.2 Příklad: Nyní budeme chtít opět vynásobit čísla 1234 a 2125, tentokrát ale změníme základ.

1) Základ $x = 100$

$$a = \underbrace{12}_{a_1}\underbrace{34}_{a_0}, \qquad b = \underbrace{21}_{b_1}\underbrace{25}_{b_0}.$$

2) Číslo $a = 1234$ vyjádříme jako polynom

$$a(x) = a_0x^0 + a_1x^1 + ... + a_{N-1}x^{N-1}.$$

Pracovat budeme s vektorem $\mathbf{a} = [\underbrace{34, 12}_{N=2}]$.

Číslo b vyjádříme pomocí polynomu

$$b(x) = b_0x^0 + b_1x^1 + ... + b_{N-1}x^{N-1}$$

a připravíme si vektor $\mathbf{b} = [\underbrace{25, 21}_{N=2}]$.

Oba vektory doplníme nulami na délku vektoru $2N$:

$$\mathbf{a'} = [\underbrace{34, 12, 0, 0}_{2N=4}], \quad \mathbf{b'} = [\underbrace{25, 21, 0, 0}_{2N=4}].$$

3) Nalezneme FFT obrazy **A', B'** vektorů **a'** a **b'**:

$\mathbf{A'} = \text{FFT}(\mathbf{a'}) =$
$[46.0000 + 0.0000i, 4.0000 - 12.0000i, 22.0000 + 0.0000i, 34.0000 + 12.0000i]$,

$\mathbf{B'} = \text{FFT}(\mathbf{b'}) =$
$[46.0000 + 0.0000i, 25.0000 - 21.0000i, 4.0000 + 0.0000i, 25.0000 + 21.0000i]$.

4) Násobíme obrazy po složkách

$\mathbf{C'} = 1000 * [2.1160 + 0.0000i, 0.5980 - 1.0140i, 0.0880 + 0.0000i, 0.5980 + 1.0140i]$.

5) Zpětnou transformací získáme výsledný vektor délky $2N$

$\mathbf{c'} = \text{IFFT}(\mathbf{C'}) = [\underbrace{850}_{c_0}, \underbrace{1014}_{c_1}, \underbrace{252}_{c_2}, \underbrace{0}_{c_3}]$

a víme, že základ $x = 100$, tedy

$$c(x) = c_0x^0 + c_1x^1 + ... + c_{2N-1}x^{2N-1} =$$

$= 850.100^0 + 1014.100^1 + 252.100^2 + 0.100^3 = 850 + 101400 + 2520000 + 0 = \underline{2622250}$

4.3.6.3 Příklad: A jak bude vypadat násobení čísel 1234 a 2125 při základu 2, si ukážeme v tomto příkladě:

1) Základ $x = 2$.

2) Číslo $a = 1234$ vyjádříme jako polynom

$$a(x) = a_0x^0 + a_1x^1 + ... + a_{N-1}x^{N-1}.$$

$1234 = 2^{10} + 2^7 + 2^6 + 2^4 + 2^1 =$
$= 1.2^{10} + 0.2^9 + 0.2^8 + 1.2^7 + 1.2^6 + 0.2^5 + 1.2^4 + 0.2^3 + 0.2^2 + 1.2^1 + 0.2^0$

Pracovat budeme s vektorem $\mathbf{a} = [\underbrace{0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1}_{N=11}]$.

Číslo $b = 2125$ vyjádříme pomocí polynomu

$$b(x) = b_0x^0 + b_1x^1 + ... + b_{N-1}x^{N-1}.$$

$2125 = 2^{11} + 2^6 + 2^3 + 2^2 + 2^0$

a připravíme si vektor $\mathbf{b} = [\underbrace{1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1}_{N=12}]$.

Oba vektory doplníme nulami na délku vektoru $2N = 24$, počítáno z vektoru o více prvcích:

a' $= [\underbrace{0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}_{2N=24}]$,

b' $= [\underbrace{1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}_{2N=24}]$.

3) Nalezneme FFT obrazy **A', B'** vektorů **a'** a **b'**:

A' $= \mathrm{FFT}(\mathbf{a'}) = [5.0000 + 0.0000i, 0.3411 - 3.5908i, ..., 0.3411 + 3.5908i]$

B' $= \mathrm{FFT}(\mathbf{b'}) = [5.0000 + 0.0000i, 1.6072 - 2.4659i, ..., 1.6072 + 2.4659i]$

4) Vynásobením obrazů po složkách získáme

C' $= [25.0000 + 0.0000i, -8.3064 - 6.6122i, ..., -8.3064 + 6.6122i]$

5) A zpětnou transformací obdržíme výsledný vektor délky $2N$

c' $= \mathrm{IFFT}(\mathbf{C'}) = [\underbrace{0, 1, 0, 1, 2, 0, 2, 3, 1, 2, 3, 0, 3, 2, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0}_{c_0, c_1, ..., c_{23}}]$

a při základu $x = 2$ dostaneme

$$c(x) = c_0x^0 + c_1x^1 + ... + c_{2N-1}x^{2N-1} =$$

$= 0.2^0+1.2^1+0.2^2+1.2^3+2.2^4+0.2^5+2.2^6+3.2^7+1.2^8+2.2^9+3.2^{10}+0.2^{11}+3.2^{12}+2.2^{13}+0.2^{14}+$
$1.2^{15}+1.2^{16}+1.2^{17}+1.2^{18}+0.2^{19}+0.2^{20}+1.2^{21}+0.2^{22}+0.2^{23} = 2+8+2.16+2.64+3.128+256+$
$2.512+3.1024+3.4096+2*8192+32768+65536+131072+262144+2097152 = \underline{2622250}$

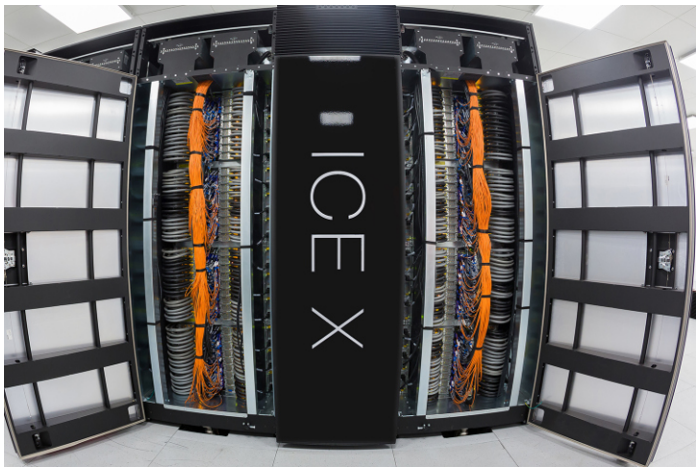
5 Numerické experimenty

V této části práce se seznámíme s testováním a porovnáváním vybraných FFT metod a budeme zjišťovat, pro jak velká N je FFT násobení rychlejší než klasické násobení a kolik času ušetří. Budeme provádět numerické experimenty v závislosti na velikosti (délce) vstupního vektoru a základu, který si zvolíme pro reprezentaci celého čísla. Cílem je si ukázat, jak který parametr ovlivňuje dobu běhu vybraného algoritmu.

Nejprve si stručně představíme superpočítač Salomon, na němž jsou experimenty prováděny.

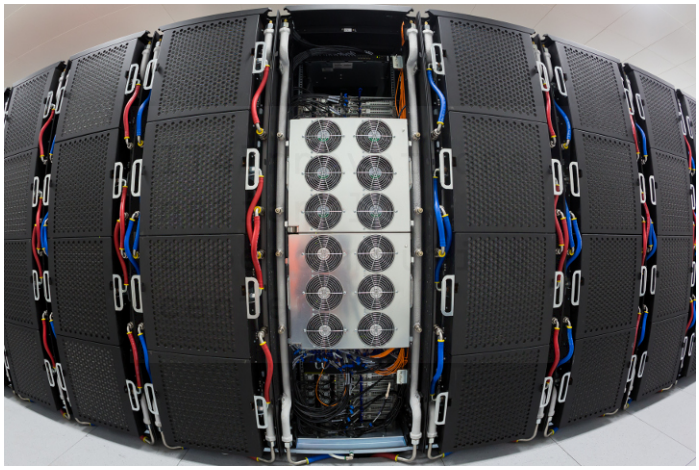
5.1 Superpočítač Salomon

Numerické experimenty týkající násobení obrovských čísel byly prováděny na superpočítačovém klastru Salomon. Klastr se skládá z 1008 výpočetních uzlů s celkovou kapacitou 24192 výpočetních jader a 129 TB paměti RAM. Výkonu je dosahováno 2Pflop/s. Každý výpočetní uzel je výkonný počítač se 32-bitovým nebo 64-bitovým operačním systémem se 24 jádry (každý server je vybaven dvěma 12tijádrovými procesory Intel Xeon) a minimálně 128 GB paměti RAM. Servery jsou propojeny vysokorychlostní sítí. Každý z nich disponuje procesorem Intel Xeon E5-2680v3. Klastr tvoří 576 uzlů bez akcelerátorů a 432 uzlů vybavených MIC akcelerátory Intel Xeon Phi.



Obrázek 19: Výpočetní uzly bez akcelerátorů

Na klastru Salomon běží operační systém Linux CentOS, který je kompatibilní s rodinou Linux RedHat.



Obrázek 20: Výpočetní uzly s MIC akcelerátorem

Obrázek 21: Chlazení výpočetních uzlů s MIC akcelerátory

5.2 Testování vlastních sekvenčních implementací

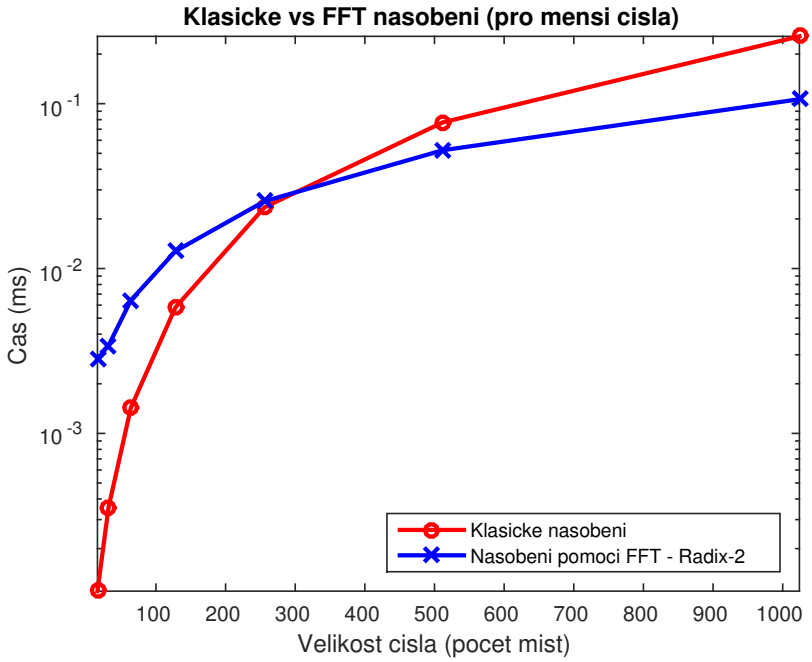
Pro numerické experimenty jsem se rozhodla si vybrat následující metody:

Radix-2, Radix-4 a Split-Radix, vše je zastoupeno jako rekurzivní i iterativní verze.

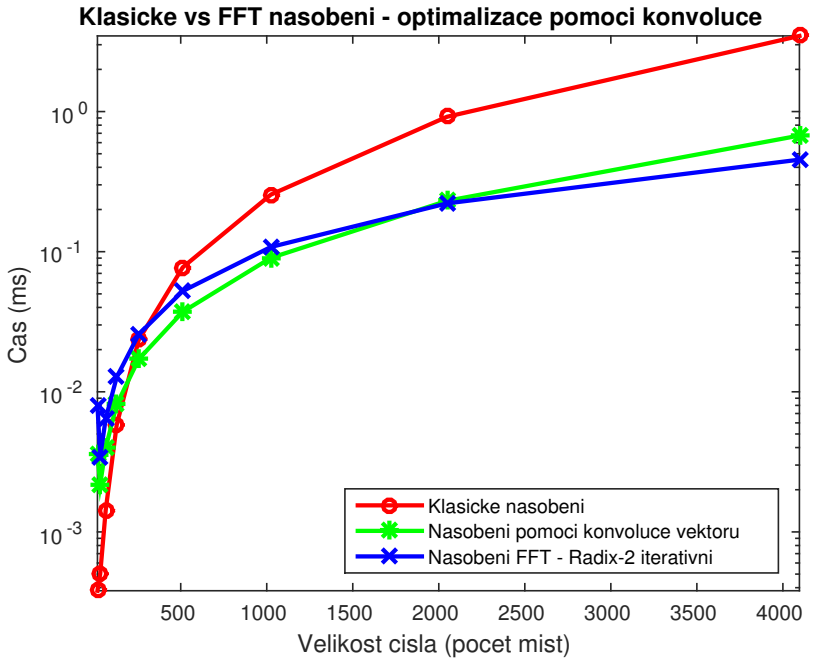
Pro snažší orientaci v časové náročnosti zpracování výsledků nejprve porovnáme FFT metodu prezentovanou Radixem-2 s klasickým násobením (Obrázek 22).

Klasické násobení lze použít zhruba do řádu 1000 pozic, dále použijeme násobení pomocí konvoluce vektorů, s nimiž lze pracovat v neomezené délce, a opět srovnáme výsledky měření (Obrázek 23).

Vidíme, že klasické násobení užívané pro "menší"čísla je rychlejší přibližně do řádu tříset pozic. Přesto, že pro větší čísla použijeme jiný způsob klasického násobení (pomocí vektorové konvo-



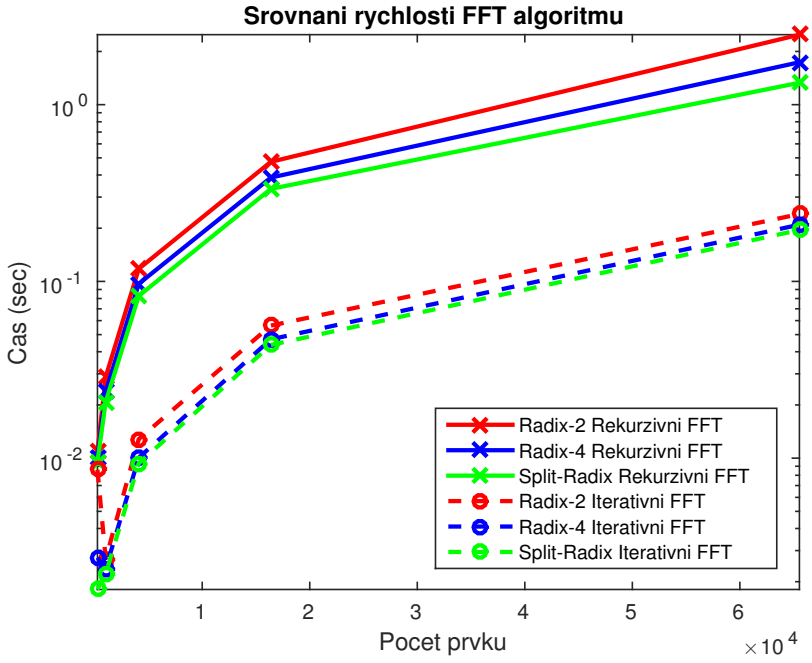
Obrázek 22: Násobení klasické vs FFT - menší čísla (Salomon - 1 jádro)



Obrázek 23: Násobení klasické vs FFT - velká čísla (Salomon - 1 jádro)

luce), a to z důvodu obrovských čísel, která by se jinak nevešla do paměti, rychlost zpracování dat bude významná do maximální velikosti přibližně tisíc pozic. Dále začíná být zajímavějším řešením FFT násobení.

Nyní již budeme porovnávat jednotlivé FFT metody. Testování provádíme pomocí algoritmů Radix-2, Radix-4 a Split-Radix, vše v rekurzivní i iterativní variantě. Jak jsme dříve zmínili u popisu jednotlivých metod 2, iterativní verze algoritmů jsou mnohem rychlejší, neboť zde odpadá vytváření mnoha pomocných polí, což se děje v případě rekurze při každém volání.



Obrázek 24: FFT násobení - srovnání jednotlivých metod (Salomon - 1 jádro)

Z grafu vidíme, že rychlost násobení zcela závisí na rychlosti použitého FFT algoritmu, nejrychlejší je iterativní Split-Radix, nicméně jsme u něj, stejně tak jako u Radix-4, omezeni velikosti vstupu na mocninu čtyř. Radix-2 je o něco pomalejší, ale pracuje s délkou vstupu, která odpovídá mocnině dvou. Výběr vhodného algoritmu a jeho paralelizace je klíčová pro zrychlení celého výpočtu.

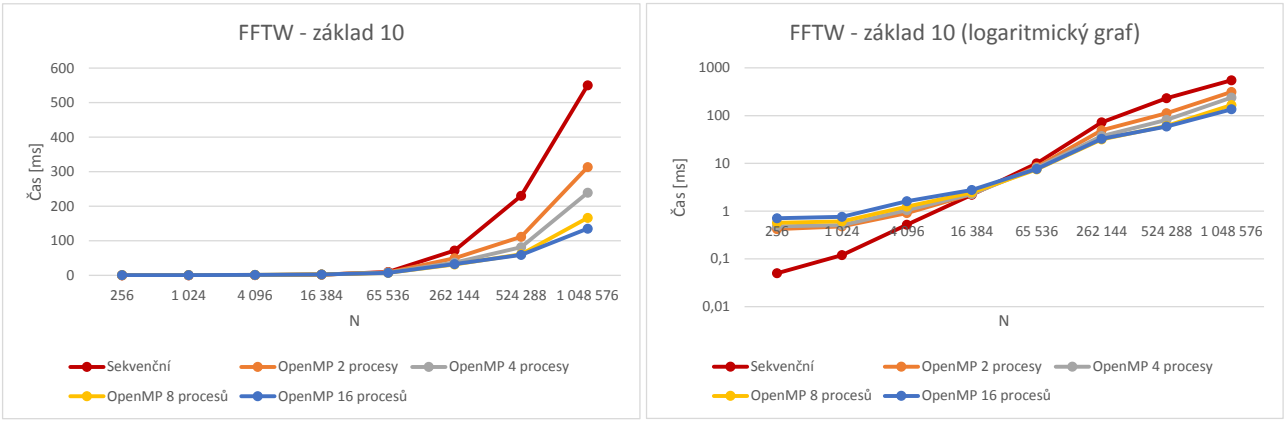
Nyní se podíváme na FFT knihovny, které mají implementované různé algoritmy pro výpočet FFT a umožňují paralelizovat FFT výpočet jak pomocí sdílené, tak distribuované paměti.

5.3 Testování paralelních implementací

Numerické experimenty byly prováděny pomocí knihoven FFTW a Intel MKL, přičemž testy byly prováděny v rámci jednoho i více uzlů o různých počtech procesů a srovnávány jsou také průběhy výpočtu obou knihoven navzájem. Obě knihovny tedy disponují možností dvou druhů paralelních výpočtů, a sice se sdílenou i s distribuovanou pamětí.

5.3.1 Numerické experimenty - sdílená paměť (OpenMP)

První testovaná knihovna je FFT. Z provedených testů je zřejmé, že knihovna má výrazně optimalizované možnosti sekvenčních algoritmů až do velikosti $N = 16384$, do níž je u paralelizace větší režie než u sekvenčního výpočtu.

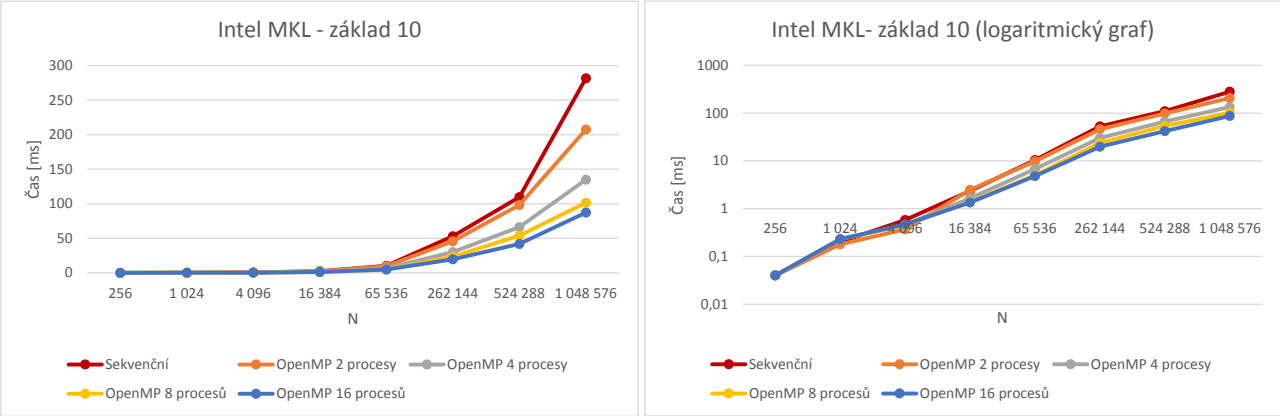


Obrázek 25: Sekvenční a paralelní implementace pomocí FFTW

FFTW 3.x		Sekvenční	Paralelní			
	N	-	2	4	8	16
Základ 10	256	0,05	0,42	0,48	0,57	0,71
	1 024	0,12	0,48	0,51	0,61	0,76
	4 096	0,52	0,91	1,05	1,24	1,61
	16 384	2,2	2,27	2,29	2,45	2,78
	65 536	10,04	8,45	8,02	7,46	7,6
	262 144	72,06	49,25	37,17	31,71	32,71
	524 288	229,97	111,82	81,38	60,68	59,36
	1 048 576	550,06	313,23	239,11	166,46	135,31
Základ 100	256	0,04	0,4	0,46	0,47	0,72
	1 024	0,07	0,4	0,47	0,55	0,71
	4 096	0,26	0,59	0,67	0,76	0,95
	16 384	0,99	1,35	1,46	1,63	2,05
	65 536	4,52	4,11	4,11	3,98	4,44
	262 144	26,08	20,39	17,61	15,8	15,13
	524 288	70,19	50,02	37,71	31,78	32,87
	1 048 576	217,32	103,26	74,51	60,85	59,58
Základ 1000	256	0,08	0,45	0,51	0,62	0,55
	1 024	0,23	2,35	2,37	3,06	4,21
	4 096	0,36	0,88	1,22	2,84	2,85
	16 384	3,08	7,06	7,11	7,2	11,54
	65 536	10,9	14,92	15,16	16,58	16,9
	262 144	55,08	94,09	100,86	110,2	99,51
	524 288	122,12	226,94	238,63	265,4	273,32
	1 048 576	196,57	152,47	125,15	108,88	102,43

Tabulka 6: Sekvenční a paralelní implementace pomocí FFTW

Knihovna Intel MKL umožňuje dosáhnout mnohem vyšších výkonů především díky vysoké optimalizaci pro hardware Intel, na kterém běží oba Ostravské superpočítače. Výsledky pro velká čísla jsou výrazně lepší než u FFTW. Z provedených testů je zřejmé, že knihovna je navržena pro paralelní výpočty.



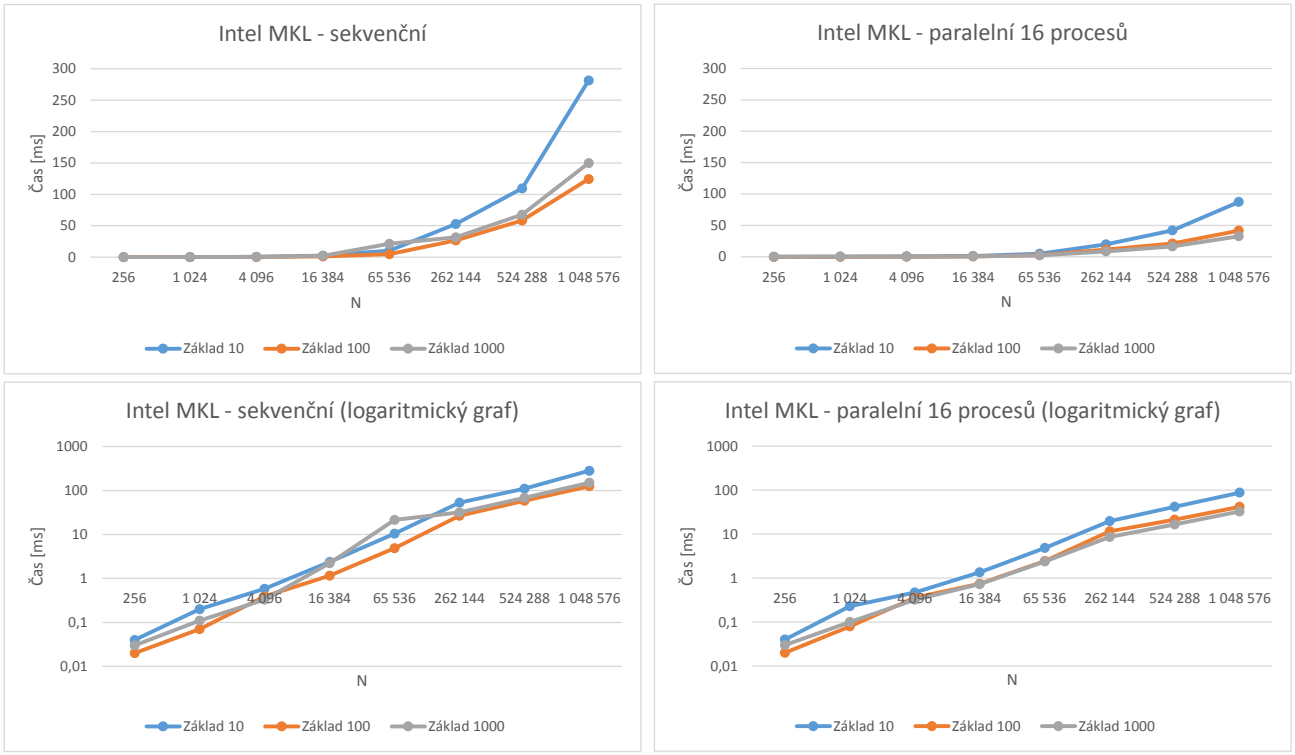
Obrázek 26: Sekvenční a paralelní implementace pomocí Intel MKL

Intel MKL		Sekvenční	Paralelní			
	N	-	2	4	8	16
Základ 10	256	0,04	0,04	0,04	0,04	0,04
	1 024	0,2	0,18	0,23	0,23	0,23
	4 096	0,58	0,37	0,46	0,45	0,47
	16 384	2,37	2,48	1,64	1,38	1,35
	65 536	10,43	9,84	6,76	4,99	4,82
	262 144	52,94	45,9	30,2	23,66	19,75
	524 288	109,58	98,23	66,46	53,94	41,86
	1 048 576	281,63	207,53	134,9	101,59	87,31
Základ 100	256	0,02	0,03	0,02	0,02	0,02
	1 024	0,07	0,08	0,07	0,08	0,08
	4 096	0,39	0,35	0,37	0,37	0,36
	16 384	1,15	0,76	0,8	0,68	0,74
	65 536	4,88	2,56	2,8	2,5	2,44
	262 144	26,77	22,59	15,07	11,5	11,61
	524 288	58,58	46,04	33,78	26,52	21,19
	1 048 576	124,54	106,16	64,77	47,52	41,67
Základ 1000	256	0,03	0,03	0,03	0,03	0,03
	1 024	0,11	0,11	0,1	0,1	0,1
	4 096	0,33	0,36	0,31	0,31	0,32
	16 384	2,23	1,13	0,84	0,73	0,73
	65 536	21,4	4,19	3,03	2,46	2,39
	262 144	31,64	21,5	13,35	10,01	8,63
	524 288	67,61	38,04	24,78	18,52	16,59
	1 048 576	150	75,36	49,19	37,11	32,67

Tabulka 7: Sekvenční a paralelní implementace pomocí Intel MKL

5.3.2 Závislost rychlosti výpočtu na použitém základu

Celý výpočet u obrovských čísel můžeme zrychlit díky závislosti na použitém základu, který se chová stejně jak při sekvenční, tak paralelní variantě. Knihovna FFTW má výsledky pro větší základy horší (ke zlepšení došlo u základu 100), u knihovny MKL však vidíme předpokládané zlepšení. Zde je třeba zdůraznit, že zatímco převod čísla ze základu 10 na základ 100, 1000 apod. je triviální, převod na jiný základ může představovat výpočetně náročnou operaci. Dále je nutné vzít v potaz, že se zvyšujícím se základem roste zaokrouhlovací chyba, které je věnována pozornost v kapitole 5.4.



Obrázek 27: Sekvenční a paralelní implementace pomocí Intel MKL

5.3.3 Numerické experimenty - distribuovaná paměť (MPI)

Při testování MPI bylo postupně využito jednoho až osmi výpočetních uzlů, s jedním až šestnácti procesy na každém z nich. Celkový počet MPI procesů pro každý test je zachycen v tabulce 8.

		Celkový počet procesů pro paralelní výpočet				
		Procesů / uzel *(mpirun -ppn 1..24)*				
		1	**2**	**4**	**8**	**16**
Uzlů	**1**	1	2	4	8	16
	2	2	4	8	16	32
	4	4	8	16	32	64
	8	8	16	32	64	128

Tabulka 8: Ilustrace počtu použitých procesů při paralelizaci

V následujících tabulkách nalezneme výsledky testování FFTW a MKL pro čísla do velikosti $N = 16384$. Obě knihovny jsou optimalizovány jiným způsobem, výsledky však nejsou příliš uspokojivé. U FFTW je zřetelná vysoká řežie 1D FFT transformace u příliš vysokého počtu MPI procesů. Obě knihovny však u výpočtu se sdílenou pamětí dosahují stejného nebo lepšího výsledku (viz předchozí kapitola 6, 7).

FFTW - MPI

N = 256

Uzlů \ Procesů / uzel	1	2	4	8	16
1	0,19	1,83	3,17	3,58	5,83
2	2,07	4,15	4,19	6,33	8,28
4	5,04	5,68	10,09	9,37	119,38
8	6,29	11,53	10,60	116,32	132,47

N = 1024

Uzlů \ Procesů / uzel	1	2	4	8	16
1	0,29	1,95	3,22	4,63	5,95
2	2,15	4,02	5,36	6,67	9,71
4	4,65	7,16	8,93	11,59	387,96
8	8,77	11,88	14,84	282,96	390,16

N = 4 096

Uzlů \ Procesů / uzel	1	2	4	8	16
1	0,87	2,79	3,56	4,94	8,28
2	2,84	4,32	5,59	8,26	10,15
4	5,54	7,34	11,19	12,21	384,00
8	8,19	14,88	18,09	289,17	908,95

N = 16 384

Uzlů \ Procesů / uzel	1	2	4	8	16
1	4,27	5,84	4,09	5,80	8,01
2	5,15	4,56	6,54	9,62	10,25
4	4,80	8,45	11,57	12,35	378,61
8	9,09	16,06	15,27	269,20	978,84

MKL - MPI

N = 256

Uzlů \ Procesů / uzel	1	2	4	8	16
1	0,11	0,09	0,17	0,20	0,13
2	0,13	0,14	0,39	0,19	0,30
4	0,12	0,30	0,19	0,28	0,58
8	0,16	0,22	0,26	0,49	0,96

N = 1024

Uzlů \ Procesů / uzel	1	2	4	8	16
1	0,40	0,23	0,25	0,22	0,28
2	0,52	0,22	0,39	0,37	0,29
4	0,28	0,36	0,42	0,28	0,58
8	0,24	0,29	0,26	0,48	0,96

N = 4 096

Uzlů \ Procesů / uzel	1	2	4	8	16
1	1,90	1,30	0,62	0,51	0,41
2	1,80	0,57	0,63	0,59	0,62
4	0,59	0,52	0,58	0,61	0,58
8	0,46	0,41	0,68	0,48	0,95

N = 16 384

Uzlů \ Procesů / uzel	1	2	4	8	16
1	27,17	16,18	8,47	4,24	2,60
2	14,95	9,23	8,03	3,26	4,05
4	9,22	9,94	2,96	2,41	2,75
8	10,10	2,47	2,05	2,43	2,16

Tabulka 9: Paralelizace středně velkých čísel o desítkovém základu

Od čísel o velikosti $N > 16384$ je zřetelné zlepšení při škálování výpočtu na více počítačů a procesorů. FFTW má obdobný problém jako u malých čísel, doba pro synchronizaci výpočtu při velkém počtu MPI procesů roste velmi rychle a nemá smysl jej dále škálovat. Knihovna MKL má tento problém zřejmě vyřešen a postupným škálováním je doba zpracování nerostoucí, dle zpracovaných výsledků klesající úměrně přiřazenému výpočetnímu výkonu.

FFTW - MPI

N = 65 536

Uzlů \ Procesů / uzel	1	2	4	8	16
1	16,74	15,26	9,35	6,88	9,63
2	15,14	10,29	6,92	9,60	14,63
4	10,57	10,31	12,04	18,51	376,24
8	9,45	14,92	20,97	308,47	972,33

N = 262 144

Uzlů \ Procesů / uzel	1	2	4	8	16
1	81,31	68,83	30,47	16,49	14,51
2	68,68	33,05	18,16	13,83	16,90
4	31,69	20,92	13,42	18,54	241,46
8	19,60	15,80	20,97	290,54	987,55

N = 524 288

Uzlů \ Procesů / uzel	1	2	4	8	16
1	194,74	115,28	71,66	31,17	20,29
2	117,07	75,05	36,04	21,78	20,64
4	71,84	38,82	22,85	21,77	411,22
8	36,17	23,31	21,88	169,65	994,14

N = 1 048 576

Uzlů \ Procesů / uzel	1	2	4	8	16
1	551,18	208,33	119,65	68,18	36,41
2	218,85	123,06	75,02	40,68	32,80
4	119,19	80,40	40,77	30,03	376,06
8	75,76	41,06	26,50	282,44	1022,17

MKL - MPI

N = 65 536

Uzlů \ Procesů / uzel	1	2	4	8	16
1	27,17	16,18	8,47	4,24	2,60
2	14,95	9,23	8,03	3,26	4,05
4	9,22	9,94	2,96	2,41	2,75
8	10,10	2,47	2,05	2,43	2,16

N = 262 144

Uzlů \ Procesů / uzel	1	2	4	8	16
1	99,14	49,36	35,08	19,23	13,78
2	49,82	32,92	23,43	21,15	10,65
4	31,83	24,44	20,51	8,56	14,09
8	18,84	20,22	6,62	5,04	4,26

N = 524 288

Uzlů \ Procesů / uzel	1	2	4	8	16
1	235,52	113,48	73,19	44,14	34,28
2	113,73	65,67	41,28	31,59	19,75
4	64,52	41,74	32,87	17,23	20,13
8	35,43	30,65	14,13	12,08	7,23

N = 1 048 576

Uzlů \ Procesů / uzel	1	2	4	8	16
1	537,09	272,87	169,26	112,35	95,04
2	248,83	116,28	83,90	57,03	78,87
4	129,97	66,87	56,88	53,44	30,62
8	70,47	49,19	48,35	19,64	32,29

Tabulka 10: Paralelizace velkých čísel o desítkovém základu

5.3.4 Srovnání MPI o základu 10 a 100 s knihovnou Intel MKL

V následujících dvou tabulkách nalezneme srovnání výsledků při použití MPI s knihovnou Intel MKL, a to násobení středně velkých čísel (11) a násobení obrovských čísel (12), obojí při základu 10 a 100.

Intel MKL - základ 10

N = 256

Uzlů \ Procesů / uzel	1	2	4	8	16
1	0,11	0,09	0,17	0,20	0,13
2	0,13	0,14	0,39	0,19	0,30
4	0,12	0,30	0,19	0,28	0,58
8	0,16	0,22	0,26	0,49	0,96

N = 1024

Uzlů \ Procesů / uzel	1	2	4	8	16
1	0,40	0,23	0,25	0,22	0,28
2	0,52	0,22	0,39	0,37	0,29
4	0,28	0,36	0,42	0,28	0,58
8	0,24	0,29	0,26	0,48	0,96

N = 4 096

Uzlů \ Procesů / uzel	1	2	4	8	16
1	1,90	1,30	0,62	0,51	0,41
2	1,80	0,57	0,63	0,59	0,62
4	0,59	0,52	0,58	0,61	0,58
8	0,46	0,41	0,68	0,48	0,95

N = 16 384

Uzlů \ Procesů / uzel	1	2	4	8	16
1	27,17	16,18	8,47	4,24	2,60
2	14,95	9,23	8,03	3,26	4,05
4	9,22	9,94	2,96	2,41	2,75
8	10,10	2,47	2,05	2,43	2,16

Intel MKL - základ 100

N = 256

Uzlů \ Procesů / uzel	1	2	4	8	16
1	0,09	0,22	0,29	0,18	0,29
2	0,13	0,14	0,39	0,19	0,30
4	0,10	0,31	0,19	0,27	0,53
8	0,35	0,23	0,26	0,43	0,78

N = 1024

Uzlů \ Procesů / uzel	1	2	4	8	16
1	0,16	0,25	0,31	0,39	0,29
2	0,16	0,25	0,31	0,39	0,29
4	0,15	0,34	0,34	0,26	0,51
8	0,43	0,36	0,26	0,44	0,77

N = 4 096

Uzlů \ Procesů / uzel	1	2	4	8	16
1	0,89	0,51	0,34	0,28	0,32
2	0,46	0,35	0,41	0,43	0,51
4	0,31	0,46	0,44	0,49	0,51
8	0,66	0,44	0,52	0,43	0,77

N = 16 384

Uzlů \ Procesů / uzel	1	2	4	8	16
1	2,92	2,03	1,27	0,65	0,51
2	2,15	1,36	0,74	0,76	0,80
4	1,13	0,76	0,73	0,72	0,82
8	1,32	0,68	0,65	0,78	0,76

Tabulka 11: Základ 10 a 100 středně velkých čísel

Intel MKL - základ 10

N = 65 536

Uzlů \ Procesů / uzel	1	2	4	8	16
1	27,17	16,18	8,47	4,24	2,60
2	14,95	9,23	8,03	3,26	4,05
4	9,22	9,94	2,96	2,41	2,75
8	10,10	2,47	2,05	2,43	2,16

N = 262 144

Uzlů \ Procesů / uzel	1	2	4	8	16
1	99,14	49,36	35,08	19,23	13,78
2	49,82	32,92	23,43	21,15	10,65
4	31,83	24,44	20,51	8,56	14,09
8	18,84	20,22	6,62	5,04	4,26

N = 524 288

Uzlů \ Procesů / uzel	1	2	4	8	16
1	235,52	113,48	73,19	44,14	34,28
2	113,73	65,67	41,28	31,59	19,75
4	64,52	41,74	32,87	17,23	20,13
8	35,43	30,65	14,13	12,08	7,23

N = 1 048 576

Uzlů \ Procesů / uzel	1	2	4	8	16
1	537,09	272,87	169,26	112,35	95,04
2	248,83	116,28	83,90	57,03	78,87
4	129,97	66,87	56,88	53,44	30,62
8	70,47	49,19	48,35	19,64	32,29

Intel MKL - základ 100

N = 65 536

Uzlů \ Procesů / uzel	1	2	4	8	16
1	11,20	6,50	3,81	2,37	1,33
2	6,60	4,89	2,46	2,17	1,66
4	4,29	2,58	1,73	1,43	1,56
8	4,41	1,40	1,40	1,33	1,36

N = 262 144

Uzlů \ Procesů / uzel	1	2	4	8	16
1	46,52	25,06	14,62	8,67	6,19
2	25,22	15,52	11,05	6,06	5,86
4	14,09	12,95	5,55	5,86	3,45
8	13,90	4,71	3,16	2,62	3,80

N = 524 288

Uzlů \ Procesů / uzel	1	2	4	8	16
1	97,26	49,51	34,30	19,25	13,91
2	51,72	30,69	23,30	20,89	10,70
4	31,72	21,52	20,21	8,59	14,01
8	18,93	20,85	6,50	4,63	4,10

N = 1 048 576

Uzlů \ Procesů / uzel	1	2	4	8	16
1	249,67	113,18	71,84	44,60	34,11
2	118,49	59,89	40,89	31,88	19,80
4	64,96	37,59	33,15	17,20	20,19
8	50,06	29,06	14,17	12,06	7,14

Tabulka 12: Základ 10 a 100 obrovských čísel

5.4 Maximální odchylka způsobená zaokrouhlováním

Při použití výpočtu pomocí FFT transformace dochází u počítání těchto obrazů k odchylce, způsobené dělením a následným zaokrouhlováním. Po vzoru článku [12] se pokusíme tuto chybu analyzovat.

Na následující ukázce je ilustrován výstup z algoritmu, který vynásobil dvě čísla reprezentované koeficienty polynomů.

```
% koeficienty polynomu reprezentujici nasobene cisla
a = [8; 9; 1; 9; 6; 0; 2; 5; 9; 9; 1; 9; 9; 4; 8; 1]
b = [4; 9; 7; 9; 6; 0; 8; 9; 6; 7; 7; 3; 6; 1; 7; 0]

% realny vysledek s chybou vzniklou delenim pri vypoctu FFT/IFFT
c = [32; 108,000000000000; 141; 180; 241; 180,000000000000; 201,000000000000; 290; 268; 361;
    459,000000000000; 421; 456; 451; 487; 572; 399; 411; 411; 302; 324; 331; 253,000000000000; 235;
    145; 127,000000000000; 118,000000000000; 42; 57,0000000000000; 6,99999999999997;
    1,42108547152020e-14; 0]

% ocekavany vysledek po zaokrouhleni
c = [32; 108; 141; 180; 241; 180; 201; 290; 268; 361; 459; 421; 456; 451; 487; 572; 399;
    411; 411; 302; 324; 331; 253; 235; 145; 127; 118; 42; 57; 7; 0; 0]

% nejvetsi chyba vnikla zaokrouhlenim koeficientu 6,99999999999997...
nejvetsi kvadraticka chyba: 3,23117426778526e-27
```

Výpis 14: Reálný a očekávaný výsledek

Po prozkoumání jednotlivých členů vektorů však zjistíme, že některé koeficienty nejsou celá čísla a budeme muset provést zaokrouhlení. Kvadrát největší odchylky způsobené zaokrouhlováním je definován vztahem:

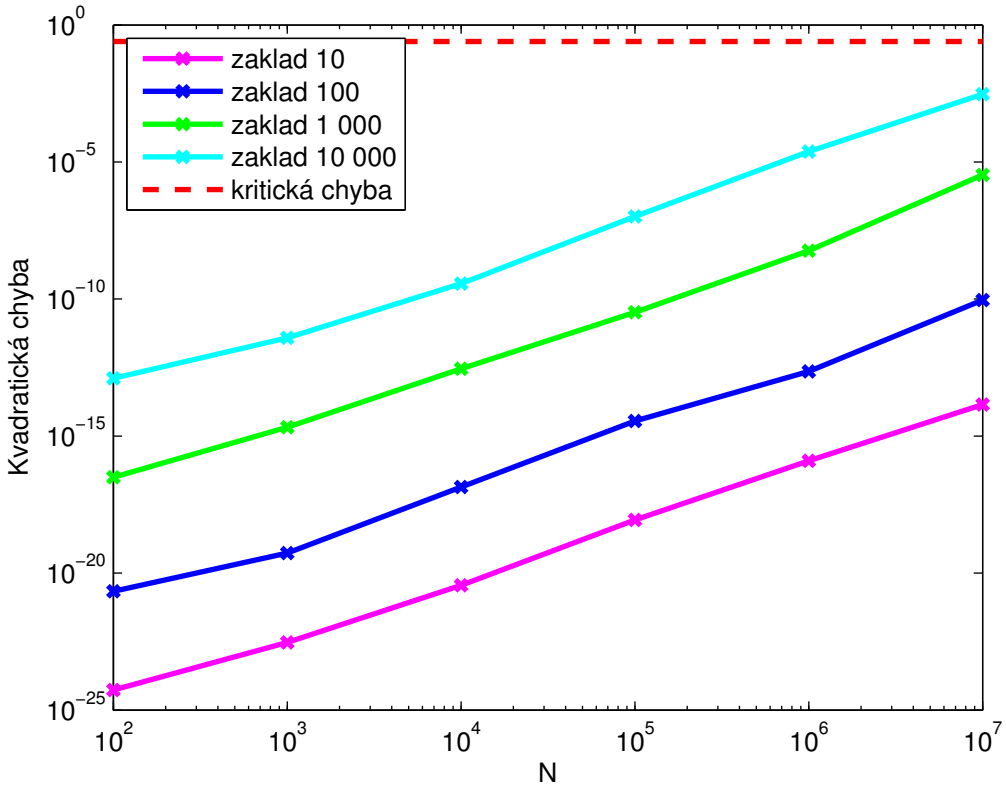
$$\text{Chyba} = (\lfloor x + 0,5 \rfloor - x)^2 .$$

Spočítáme tuto chybu pro každý koeficient polynomu a nalezneme maximální chybu. Stejným způsobem hledáme maximální chybu pro další čísla a zjišťujeme, jak se její hodnota mění v závislosti na velikosti N či zvoleném základu. Průběžné výsledky měření zaznamenáváme do tabulky 13, z níž vyplývá, že zanedbatelných[8] odchylek se dopouštíme při násobení co nejmenších čísel a malých základech, zatímco s rostoucím počtem cifer se maximální[9] chyba zvětšuje.

[8] Čím tmavší zelená barva vykresluje buňku, tím nižší hodnotu obsahuje.
[9] Zatímco největší hodnoty reprezentuje tmavě žlutá.

		N					
		100	1 000	10 000	100 000	1 000 000	10 000 000
Základ	10	5,46E-25	2,98E-23	3,60E-21	8,67E-19	1,25E-16	1,42E-14
	100	2,17E-21	5,42E-20	1,39E-17	3,55E-15	2,27E-13	9,09E-11
	1 000	3,12E-17	2,10E-15	2,88E-13	3,27E-11	5,82E-09	3,35E-06
	10 000	1,28E-13	3,81E-12	3,64E-10	1,03E-07	2,38E-05	0,00299072

Tabulka 13: Největší odchylka



Obrázek 28: Největší odchylka při různých základech

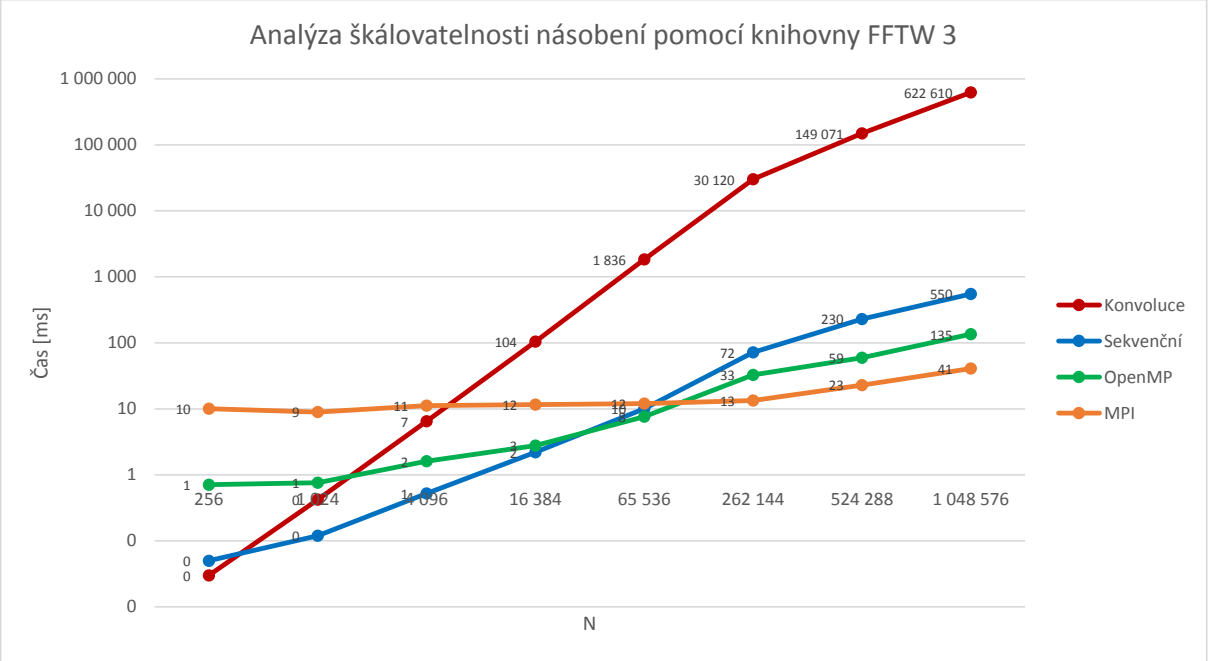
V grafu je ilustrován nárůst chyby a také její kritická hodnota $0,25$, nad kterou by již nebylo jednoznačné, zda jsme zaokrouhlení provedli správně či nikoliv. Je tedy zřejmé, že čísla, jejichž počet cifer je 10^7, mají při vysokém zvoleném základu maximální kvadratickou chybu těsně pod kritickou hranicí, tudíž u těchto a větších čísel je dobré použít nižší číselný základ, v opačném případě výpočty mohou být nekorektní.

5.5 Analýza škálovatelnosti násobení pomocí testovaných knihoven

Při násobení obrovských čísel $N \geq 100000$ s knihovnou FFTW se prokázalo, že rozložením výpočtu na více výpočetních uzlů je možné dosáhnout více než desetinásobného zrychlení ve srovnání se sekvenčním FFT násobením. U menších čísel je výhodnější využít sekvenční FFT variantu. Počet uzlů pro MPI výpočet je však nutné volit vhodně, u knihovny FFTW jsme dosáhli optimálních výsledků při kombinaci 4 uzlů se 4-mi procesy na uzel. V tabulce 14 je zachyceno zrychlení sekvenčního výpočtu pomocí FFT oproti násobení pomocí konvoluce a dále zrychlení paralelního (OpenMP a MPI) FFT oproti sekvenční variantě.

				16 vláken		4 uzly / ppn=4	
N	Konvoluce	Sekvenční		OpenMP		MPI	
	Čas [ms]	Čas [ms]	Zrychlení	Čas [ms]	Zrychlení	Čas [ms]	Zrychlení
256	0,03	0,05	*0,60*	0,71	*0,07*	10,09	*0,00*
1 024	0,42	0,12	*3,50*	0,76	*0,16*	8,93	*0,01*
4 096	6,50	0,52	*12,50*	1,61	*0,32*	11,19	*0,05*
16 384	103,98	2,20	*47,26*	2,78	*0,79*	11,57	*0,19*
65 536	1 836,19	10,04	*182,89*	7,60	*1,32*	12,04	*0,83*
262 144	30 120,00	72,06	*417,99*	32,71	*2,20*	13,42	*5,37*
524 288	149 071,07	229,97	*648,22*	59,36	*3,87*	22,85	*10,06*
1 048 576	622 609,69	550,06	*1 131,89*	135,31	*4,07*	40,77	*13,49*

Tabulka 14: Analýza škálovatelnosti násobení pomocí FFTW

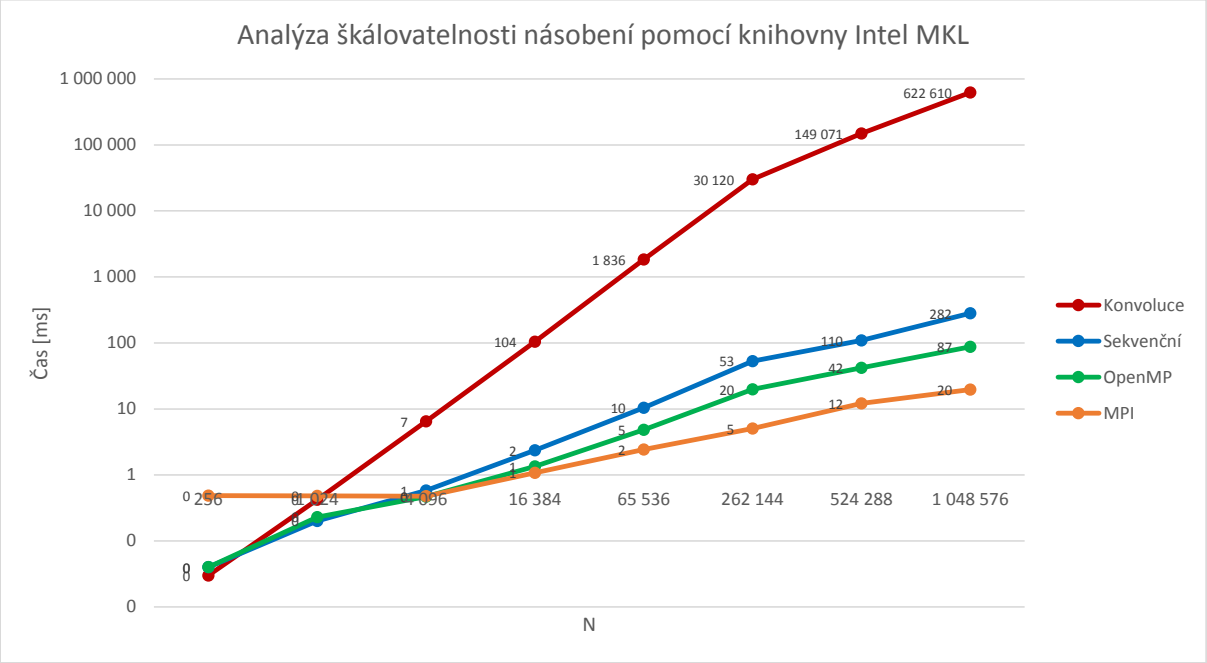


Obrázek 29: Analýza škálovatelnosti násobení pomocí FFTW

U knihovny MKL docházíme k podobnému závěru, více než desetinásobné zrychlení oproti sekvenční variantě [10] při obrovských číslech, ovšem využití paralelních metod je v tomto případě výhodné již od menších čísel. Oproti FFTW je zde dosaženo ještě vyšších výkonů (*všechny knihovny budou srovnány vzájemně v další části textu*). Optimální nastavení MPI je 8 uzlů a 8 procesů na uzel (tabulka 15).

				16 vláken		8 uzlů / ppn=8	
N	**Konvoluce**	**Sekvenční**		**OpenMP**		**MPI**	
	Čas [ms]	Čas [ms]	Zrychlení	Čas [ms]	Zrychlení	Čas [ms]	Zrychlení
256	0,03	0,04	*0,75*	0,04	*1,00*	0,49	*0,08*
1 024	0,42	0,20	*2,10*	0,23	*0,87*	0,48	*0,41*
4 096	6,50	0,58	*11,21*	0,47	*1,23*	0,48	*1,22*
16 384	103,98	2,37	*43,87*	1,35	*1,76*	1,08	*2,20*
65 536	1 836,19	10,43	*176,05*	4,82	*2,16*	2,43	*4,29*
262 144	30 120,00	52,94	*568,95*	19,75	*2,68*	5,04	*10,49*
524 288	149 071,07	109,58	*1 360,39*	41,86	*2,62*	12,08	*9,07*
1 048 576	622 609,69	281,63	*2 210,74*	87,31	*3,23*	19,64	*14,34*

Tabulka 15: Analýza škálovatelnosti násobení pomocí Intel MKL



Obrázek 30: Analýza škálovatelnosti násobení pomocí Intel MKL

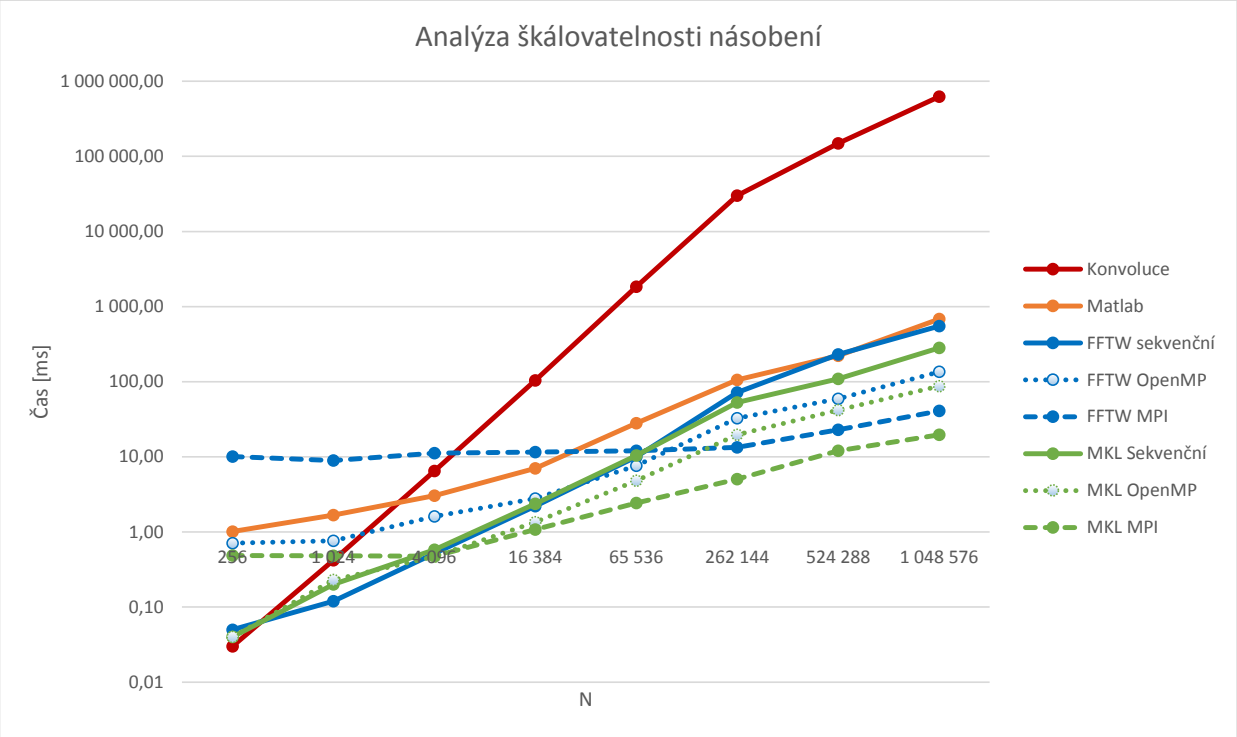
[10] Rovněž u MKL je zachyceno zrychlení sekvenčního výpočtu pomocí FFT oproti násobení pomocí konvoluce a dále zrychlení paralelního (OpenMP a MPI) FFT oproti sekvenční variantě.

5.6 Srovnání výsledků

Po shrnutí průběžných výsledků, viz tabulka 16, je prokazatelné, že u čísel přesahující velikost $N = 65536$, má smysl škálovat násobení za pomocí distribuovaných výpočtů, případně paralelizace se sdílenou pamětí. Z provedených měření vychází jako vhodná volba pro paralelní FFT knihovnu na ostravském superpočítači knihovna Intel MKL. Z pohledu přenositelnosti kódu je výhodné používat funkce FFTW a tuto knihovnu využít až při kompilaci v produkčním prostředí.

N	Konvoluce	Matlab	FFTW			Intel MKL		
	Čas [ms]	Čas [ms]	Sekvenční	OpenMP	MPI	Sekvenční	OpenMP	MPI
256	0,03	1,01	0,05	0,71	10,09	0,04	0,04	0,49
1 024	0,42	1,68	0,12	0,76	8,93	0,20	0,23	0,48
4 096	6,50	3,05	0,52	1,61	11,19	0,58	0,47	0,48
16 384	103,98	7,03	2,20	2,78	11,57	2,37	1,35	1,08
65 536	1 836,19	27,93	10,04	7,60	12,04	10,43	4,82	2,43
262 144	30 120,00	105,50	72,06	32,71	13,42	52,94	19,75	5,04
524 288	149 071,07	221,92	229,97	59,36	22,85	109,58	41,86	12,08
1 048 576	622 609,69	681,63	550,06	135,31	40,77	281,63	87,31	19,64

Tabulka 16: Analýza škálovatelnosti násobení



Obrázek 31: Analýza škálovatelnosti násobení

6 Zhodnocení výsledků

V první části práce byl čtenář seznámen se základní charakteristikou FFT metod s časovou složitostí $O(N \log N)$, díky níž jsme schopni dosahovat výsledků při násobení obrovských celých čísel mnohem efektivněji, než při klasickém násobení s časovou složitostí $O(N^2)$, jednotlivými variantami i principy, na nichž jsou založeny. Rovněž byly představeny algoritmy Radix-2, Radix-4 a Split-Radix, díky nimž bylo možné porovnávat časovou náročnost těchto FFT metod vzájemně mezi sebou.

Stručný přehled FFT knihoven a možností, jakými dané knihovny disponují, jsou prezentovány v další kapitole.

Vylíčili jsme techniku převodu celého čísla na polynom a způsob, jak pomocí FFT dvě celá čísla vynásobíme. Čtenář byl seznámen s možnostmi násobení a důvody, proč se snažíme klasické násobení nahradit pomocí alternativ. Grafy a tabulky s naměřenými hodnotami mu umožnily získat přehled, v jaké fázi velikosti N je vhodná která varianta násobení.

Provedli jsme řadu experimentů na suporpočítači Salomon, také s knihovnami FFTW a Intel MKL, kde jsme porovnávali násobení celých čísel o různém počtu cifer, dále o různých základech, a to jak sekvenční výpočty, tak paralelní s využitím OpenMP i MPI. Ukázalo se, že u obrovských čísel má smysl tuto metodu paralelizovat, čímž umožníme efektivně využít dostupný hardware ke zrychlení celého výpočtu.

Literatura

[1] RAO, K, D KIM a J HWANG. *Fast Fourier transform: algorithms and applications.* New York: Springer, c2010. ISBN 1402066295.

[2] CHU, Eleanor Chin-hwa a Alan. GEORGE. Inside the FFT black box: serial and parallel fast Fourier transform algorithms. Boca Raton, Fla.: CRC Press, c2000. ISBN 0849302706.

[3] WILLERS, Michael. *Algebra bez (m)učení: od arabských matematiků k tajným šifrám: matematika v každodenním životě : fascinující čísla a rovnice.* 1. vyd. Praha: Grada, 2012. ISBN 978-80-247-4123-9.

[4] http://www.cis.rit.edu/class/simg716/Gauss_History_FFT.pdf

[5] https://web.njit.edu/ jiang/math614/cooley-tukey.pdf

[6] https://en.wikipedia.org/wiki/Cooley–Tukey_FFT_algorithm

[7] http://www.webabode.com/articles/Parallel%20FFT%20implementations.pdf

[8] http://www.cypress.com/file/55401/download

[9] http://www.fizyka.umk.pl/ daras/ps/fs/2/FFT%20IFFT.pdf

[10] http://www.mathworks.com/help/signal/ref/digitrevorder.html

[11] https://en.wikipedia.org/wiki/Discrete_Hartley_transform

[12] http://www.cs.rug.nl/ ando/pdfs/Ando_Emerencia_multiplying_huge_integers_using_fourier_trans

[13] http://mi21.vsb.cz/sites/mi21.vsb.cz/files/unit/funkce_komplexni_promenne.pdf

[14] http://mi21.vsb.cz/sites/mi21.vsb.cz/files/unit/diskretni_transformace.pdf

[15] http://turing.cz/ tom/efa/texty/44-fft.pdf

[16] http://www.matematyka.estranky.cz/clanky/historie-matematiky.html

[17] https://www.pf.jcu.cz/stru/katedry/m/knihy/DejinyM.pdf

[18] http://bimbo.fjfi.cvut.cz/ soc/Nasobeni_papir/egyptske_nasobeni.html

[19] http://www.antickysvet.cz

[20] https://kmlinux.fjfi.cvut.cz/ balkolub/papers/PokrokyNasobeni.pdf